이진 탐색 트리의 종류 (자동 밸런스)

AVL
2-3
2-3-4
B tree
B+ tree
B** tree
...
RB tree

위키피디아

RB tree 삽입 룰

1

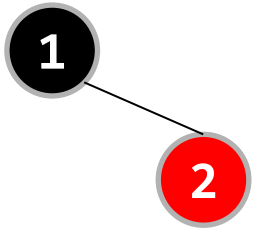
1. 새로운 노드는 빨강이다.

RB tree 삽입 룰

1

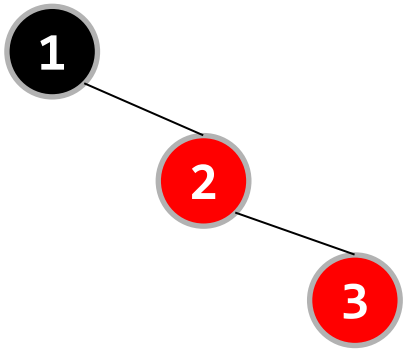
1. root 언제나 검정이다.
2. 새로운 노드는 빨강이다.

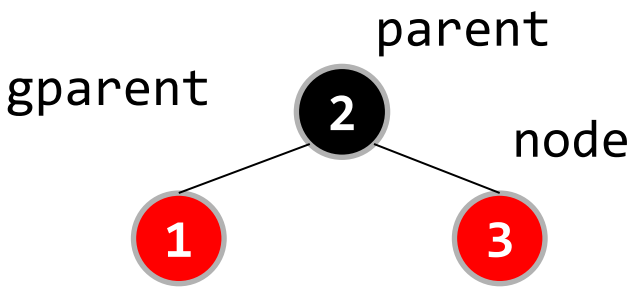
RB tree 삽입 룰



1. root 언제나 검정이다.
2. 새로운 노드는 빨강이다.

RB tree 삽입 룰

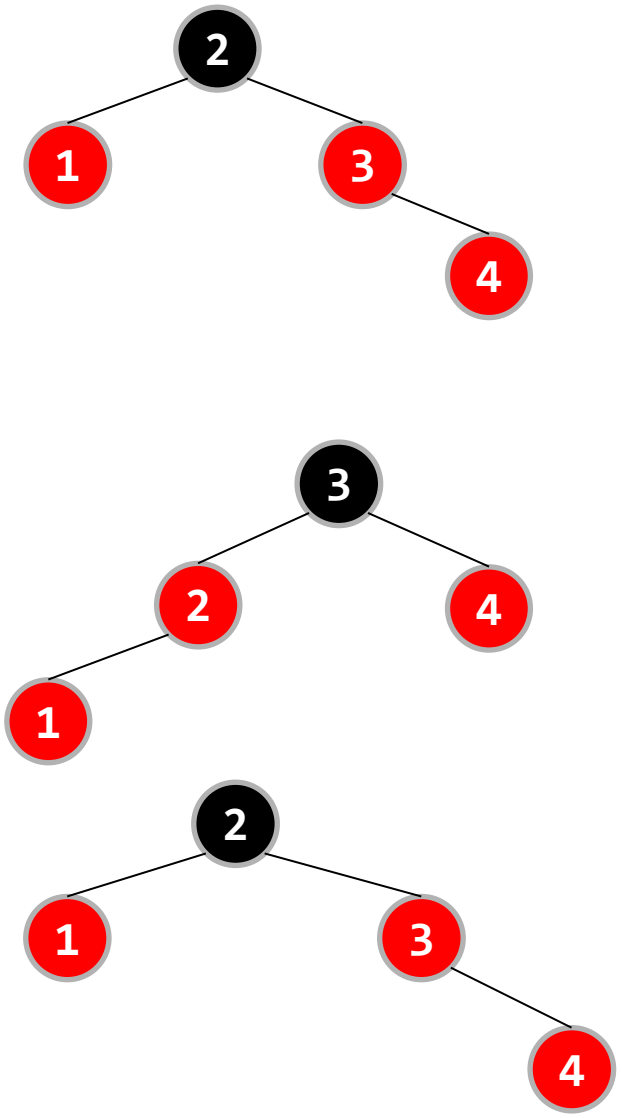




1. root 언제나 검정이다.
2. 새로운 노드는 빨강이다.
3. 부모가 빨강이면 회전한다.

R-R 경사
color flip
parent => BLACK
gparent => RED
rotate_left(gparent)

RB tree 삽입 룰

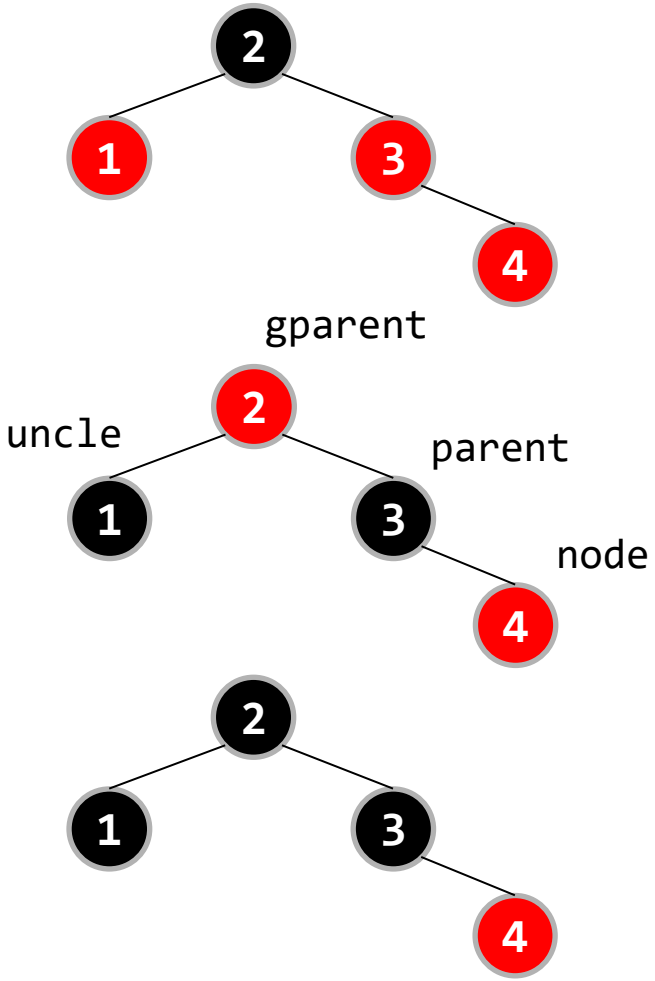


1. root 언제나 검정이다.
2. 새로운 노드는 빨강이다.
3. 부모가 빨강이면 회전한다.

R-R 경사
color flip
parent => BLACK
gparent => RED
rotate_left(gparent)

L-L 경사
color flip
parent => BLACK
gparent => RED
rotate_right(gparent)

RB tree 삽입 룰

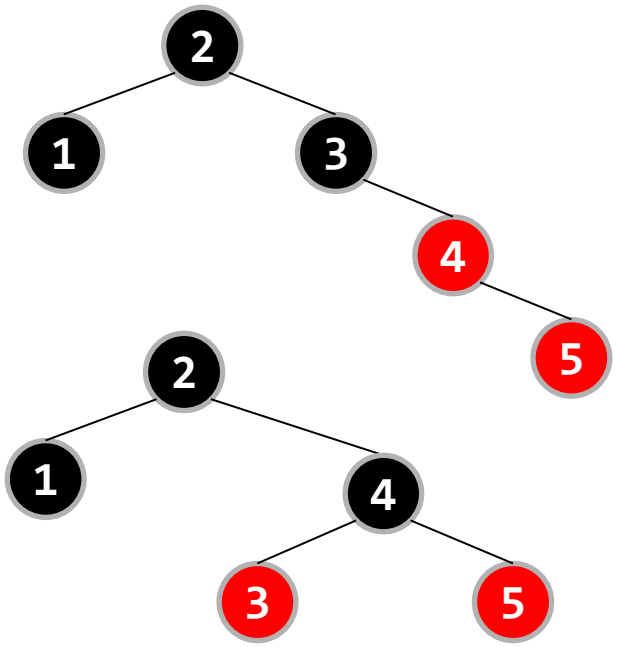


1. root 언제나 검정이다.
2. 새로운 노드는 빨강이다.
3. 부모도 빨강이고 삼촌도 빨강
   color flip 만 한다.
   uncle, parent => BLACK
   gparent => RED
4. 부모가 빨강이면 회전한다.

R-R 경사
color flip
parent => BLACK
gparent => RED
rotate_left(gparent)

L-L 경사
color flip
parent => BLACK
gparent => RED
rotate_right(gparent)

RB tree 삽입 룰

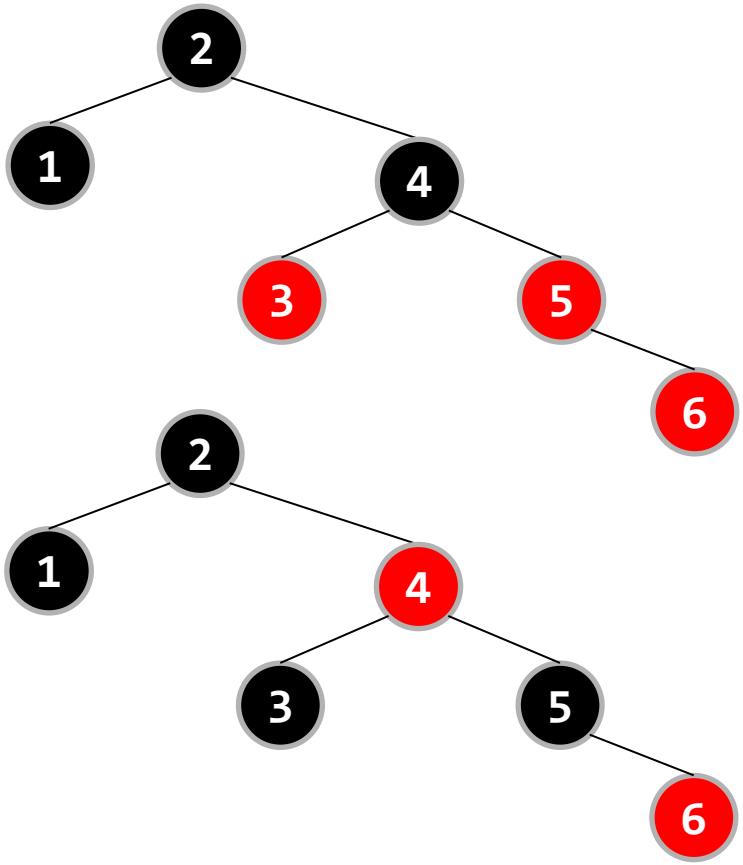


0. 밸런스는 루트로 부터 모든 단말 노드로 가는 경로의 검정 노드의 수가 같으면 맞는 것으로 간주한다.
1. root 언제나 검정이다.
2. 새로운 노드는 빨강이다.
3. 부모도 빨강이고 삼촌도 빨강
   color flip 만 한다.
   uncle, parent => BLACK
   gparent => RED
4. 부모가 빨강이면 회전한다.

R-R 경사
color flip
parent => BLACK
gparent => RED
rotate_left(gparent)

L-L 경사
color flip
parent => BLACK
gparent => RED
rotate_right(gparent)

RB tree 삽입 룰

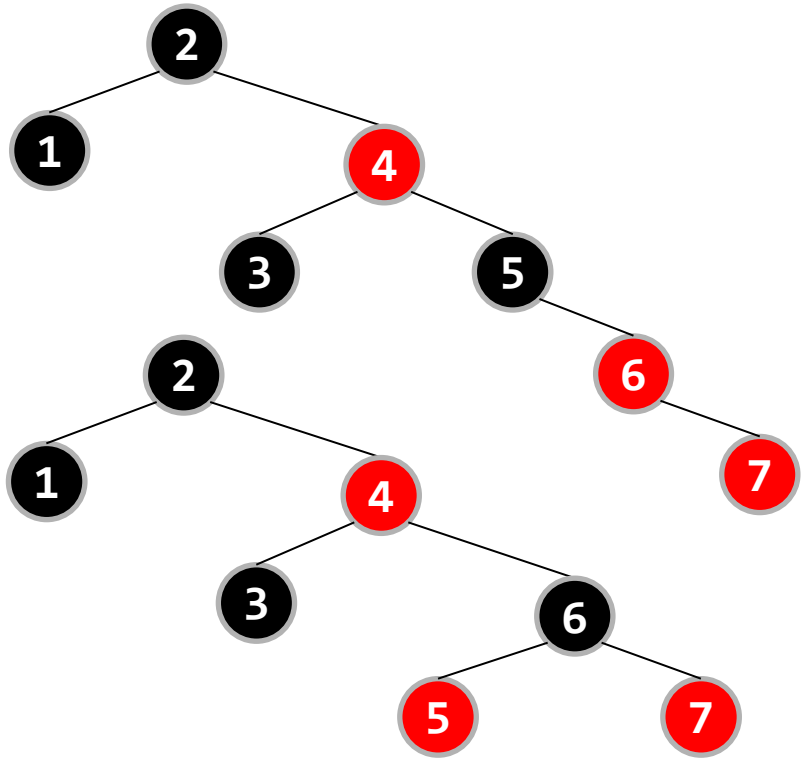


0. 밸런스는 루트로 부터 모든 단말 노드로 가는 경로의 검정 노드의 수가 같으면 맞는 것으로 간주한다.
1. root 언제나 검정이다.
2. 새로운 노드는 빨강이다.
3. 부모도 빨강이고 삼촌도 빨강
   color flip 만 한다.
   uncle, parent => BLACK
   gparent => RED
4. 부모가 빨강이면 회전한다.

R-R 경사
color flip
parent => BLACK
gparent => RED
rotate_left(gparent)

L-L 경사
color flip
parent => BLACK
gparent => RED
rotate_right(gparent)

RB tree 삽입 룰

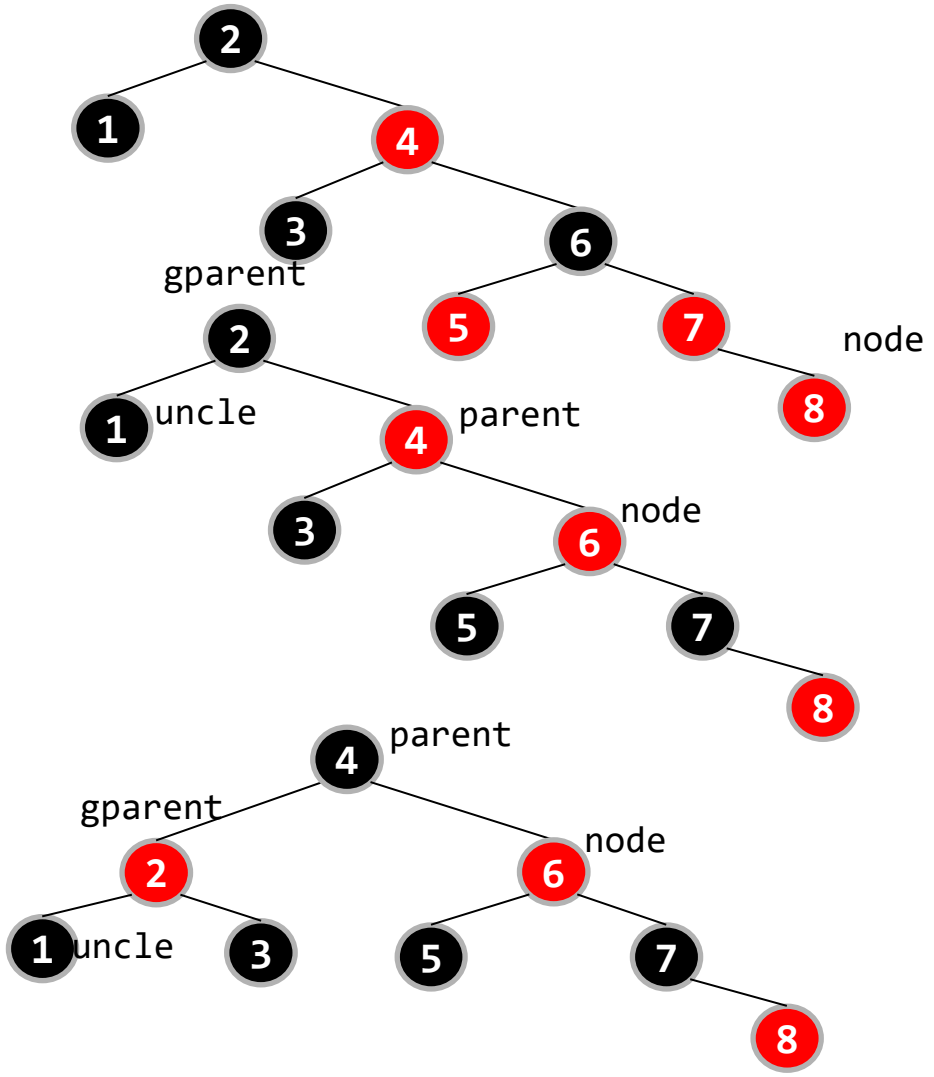


0. 밸런스는 루트로 부터 모든 단말 노드로 가는 경로의 검정 노드의 수가 같으면 맞는 것으로 간주한다.
1. root 언제나 검정이다.
2. 새로운 노드는 빨강이다.
3. 부모도 빨강이고 삼촌도 빨강
   color flip 만 한다.
   uncle, parent => BLACK
   gparent => RED
4. 부모가 빨강이면 회전한다.

R-R 경사
color flip
parent => BLACK
gparent => RED
rotate_left(gparent)

L-L 경사
color flip
parent => BLACK
gparent => RED
rotate_right(gparent)

RB tree 삽입 룰

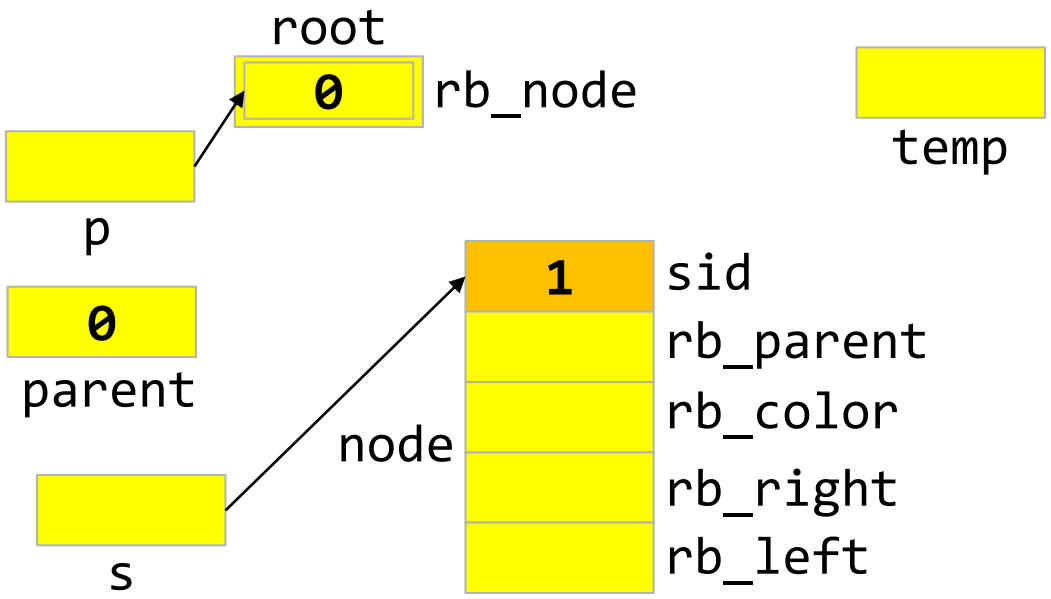


0. 밸런스는 루트로 부터 모든 단말 노드로 가는 경로의 검정 노드의 수가 같으면 맞는 것으로 간주한다.
1. root 언제나 검정이다.
2. 새로운 노드는 빨강이다.
3. 부모도 빨강이고 삼촌도 빨강
   color flip 만 한다.
   uncle, parent => BLACK
   gparent => RED
4. 부모가 빨강이면 회전한다.

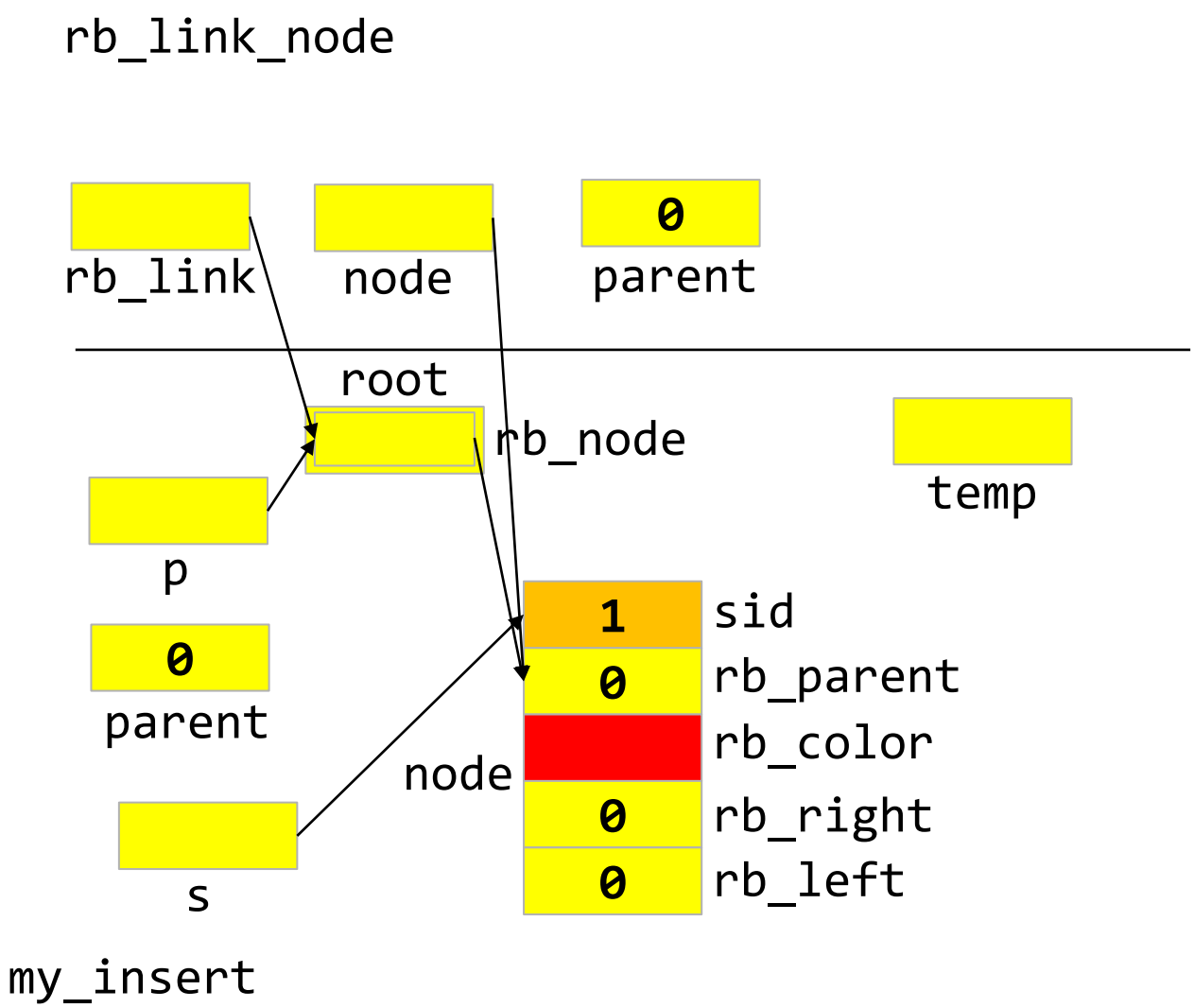
R-R 경사
color flip
parent => BLACK
gparent => RED
rotate_left(gparent)

L-L 경사
color flip
parent => BLACK
gparent => RED
rotate_right(gparent)

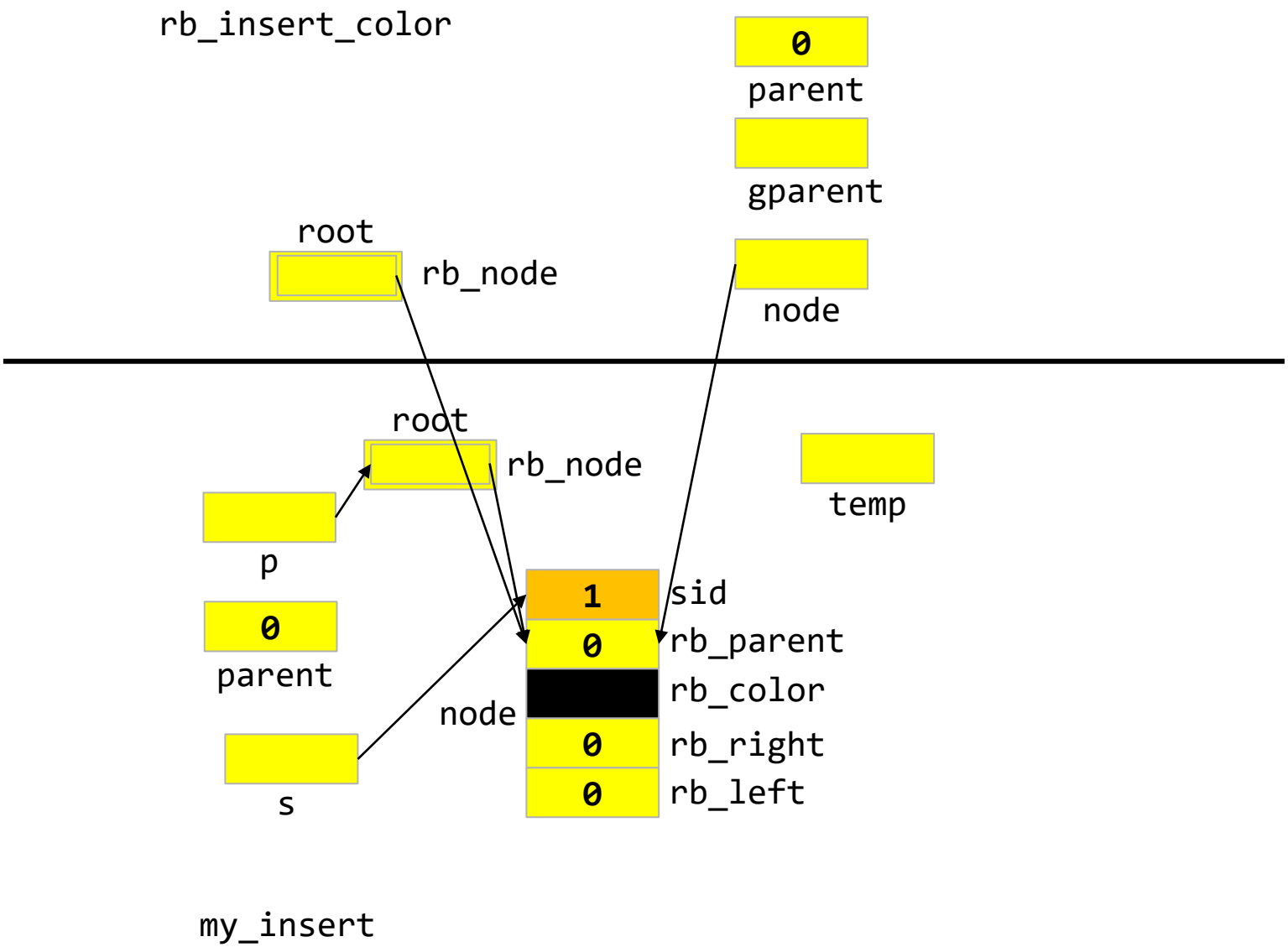
RB tree 삽입 코드 분석



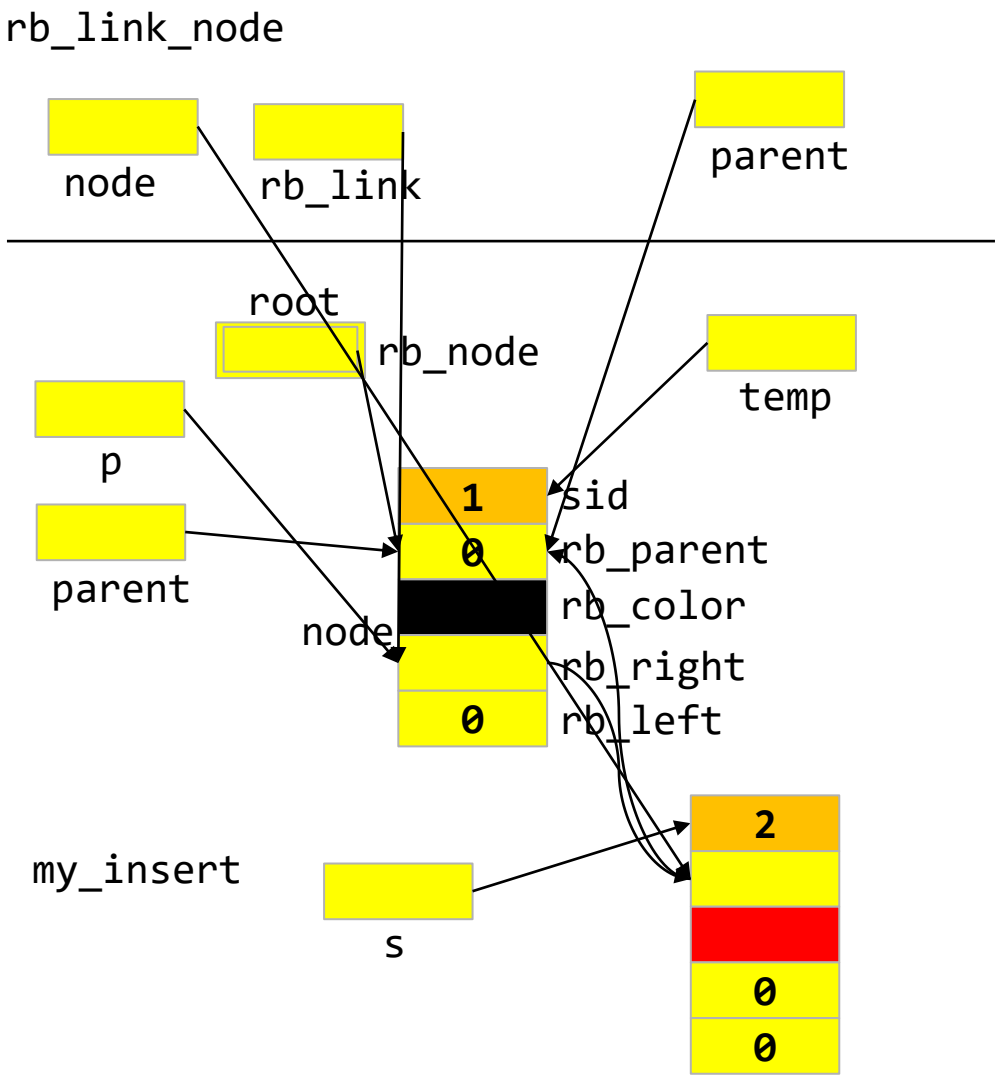
RB tree 삽입 코드 분석



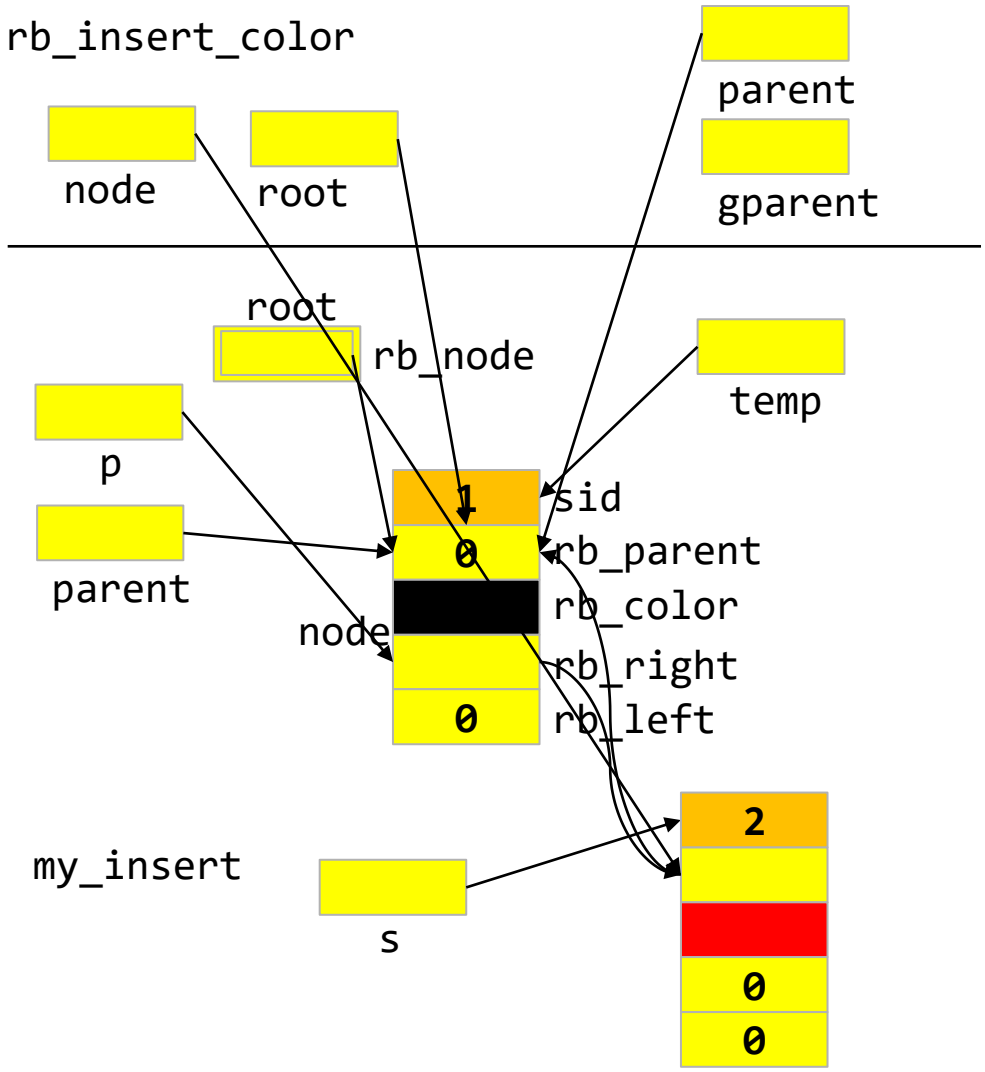
RB tree 삽입 코드 분석



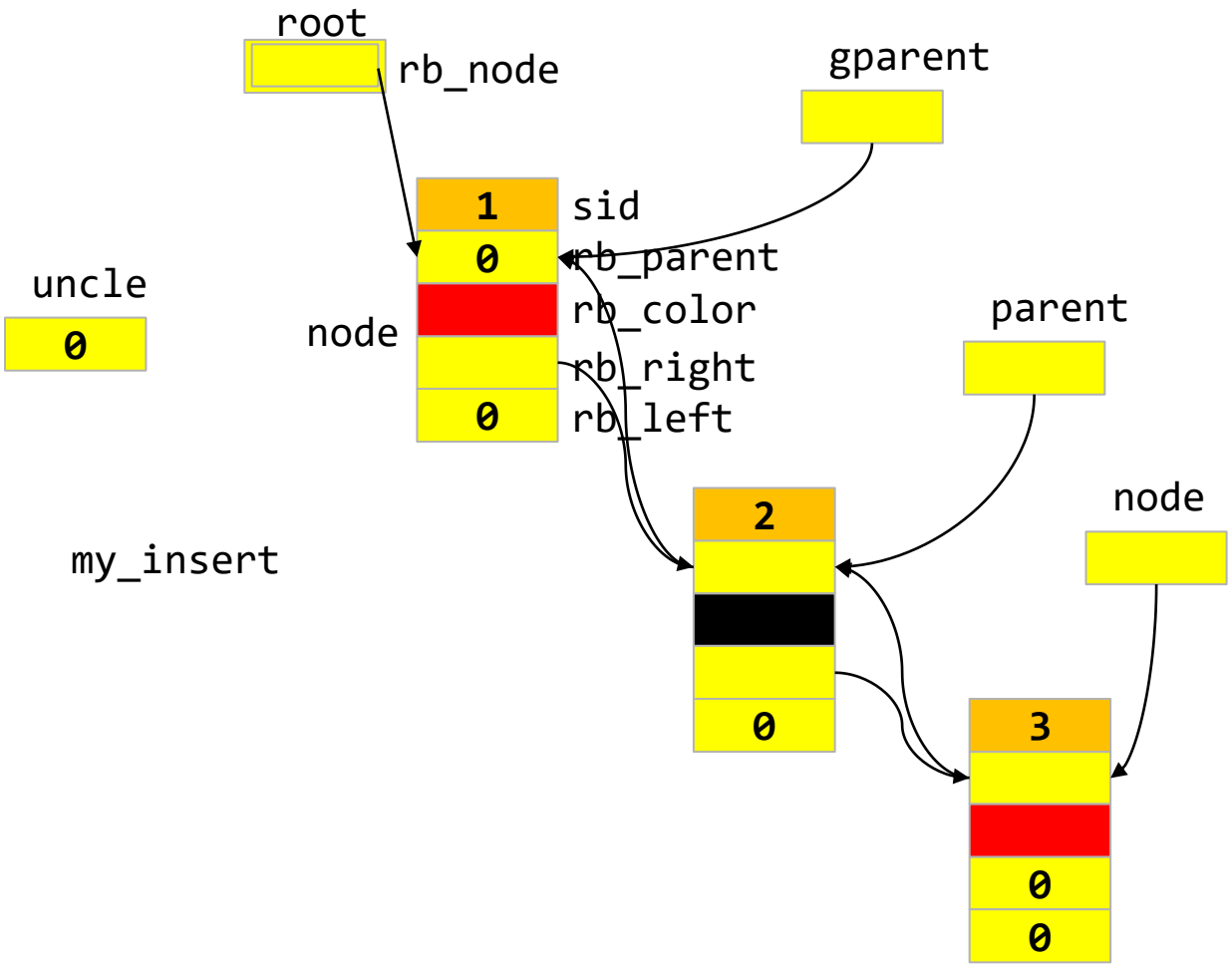
RB tree 삽입 코드 분석



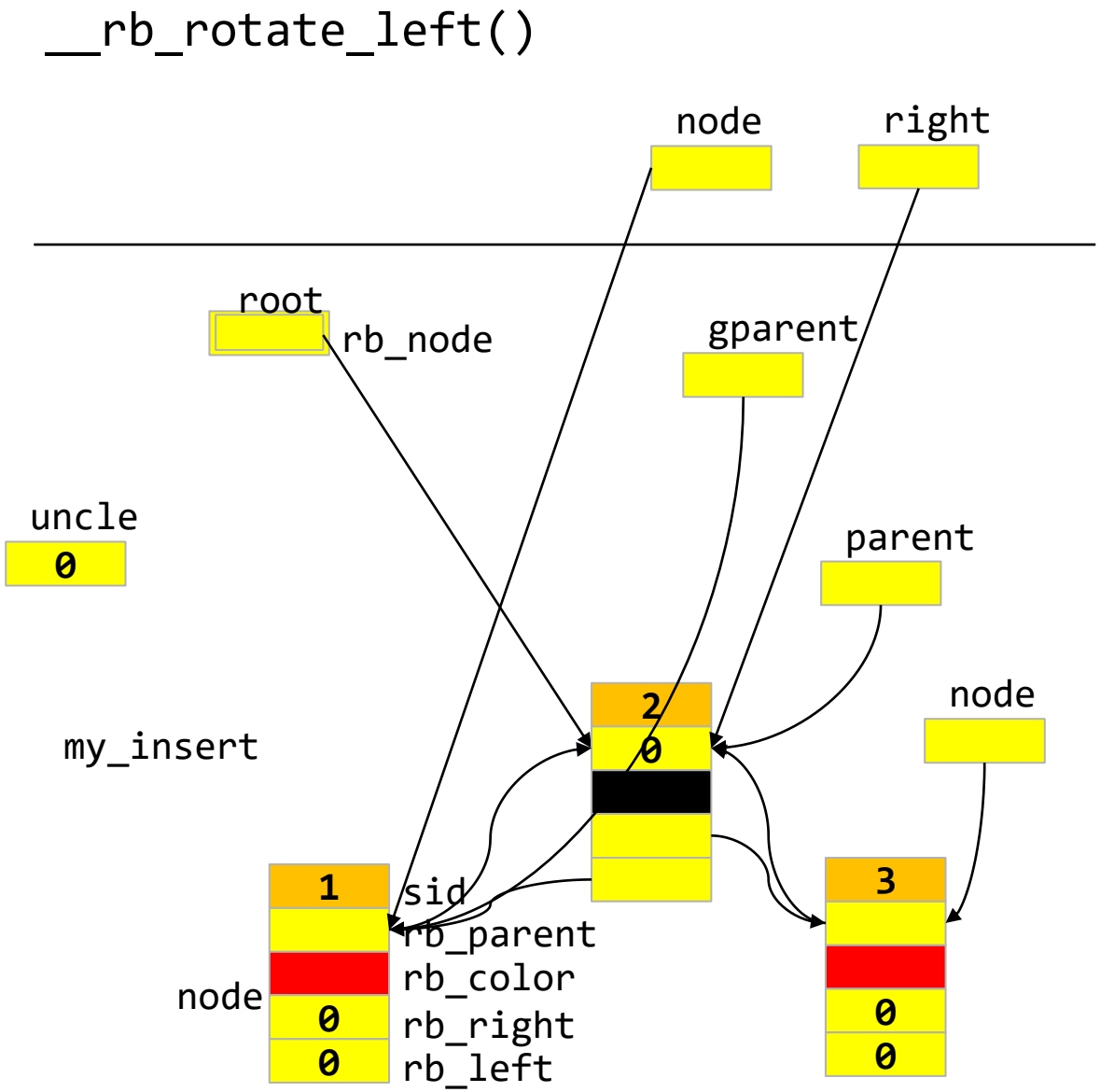
RB tree 삽입 코드 분석



RB tree 삽입 코드 분석

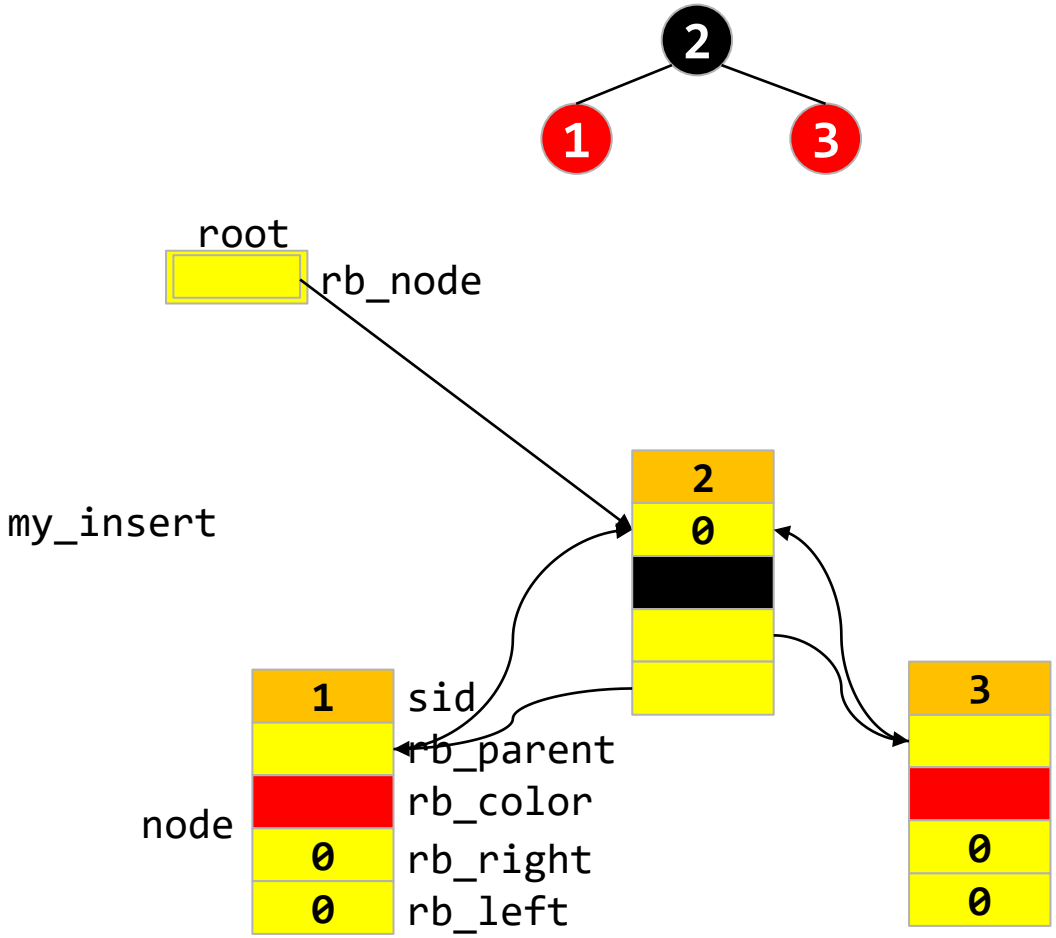


RB tree 삽입 코드 분석

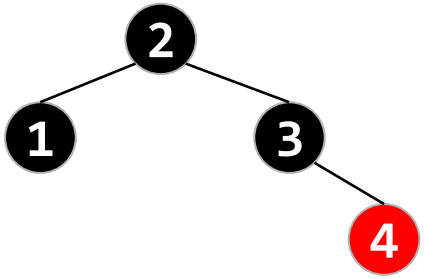


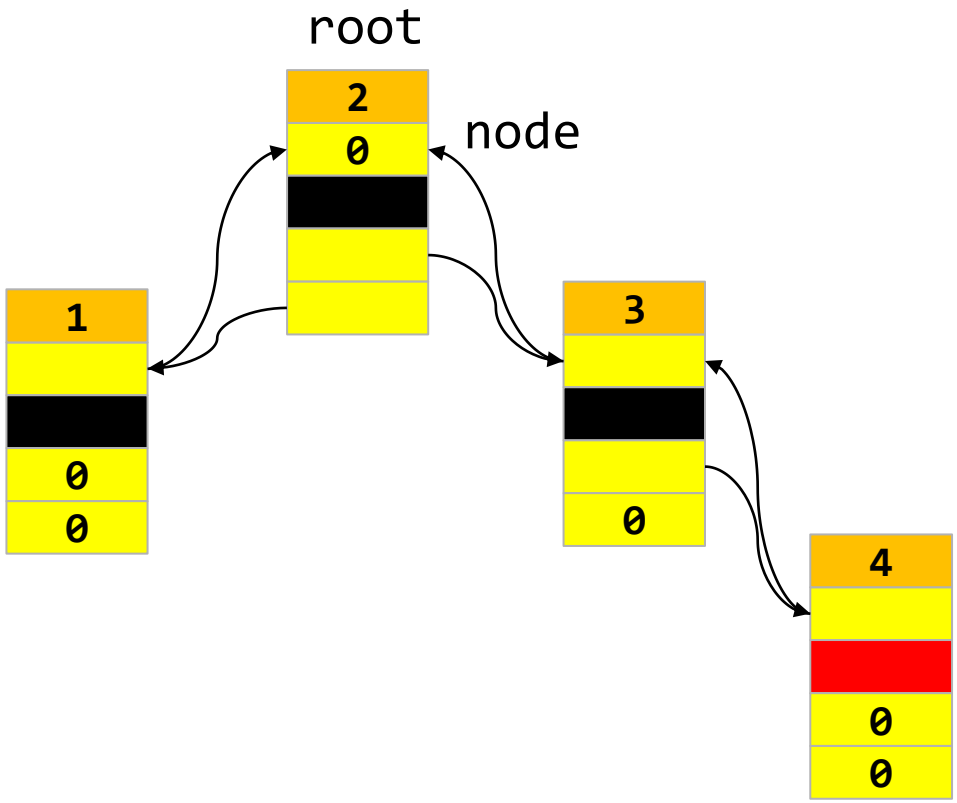
RB tree 삽입 코드 분석

__rb_rotate_left(gparent)

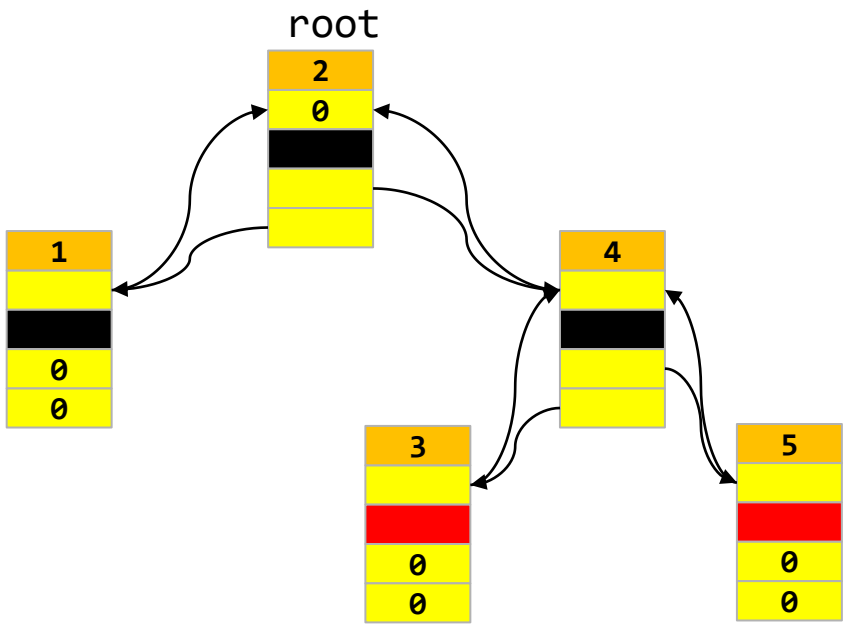


RB tree 삽입 코드 분석

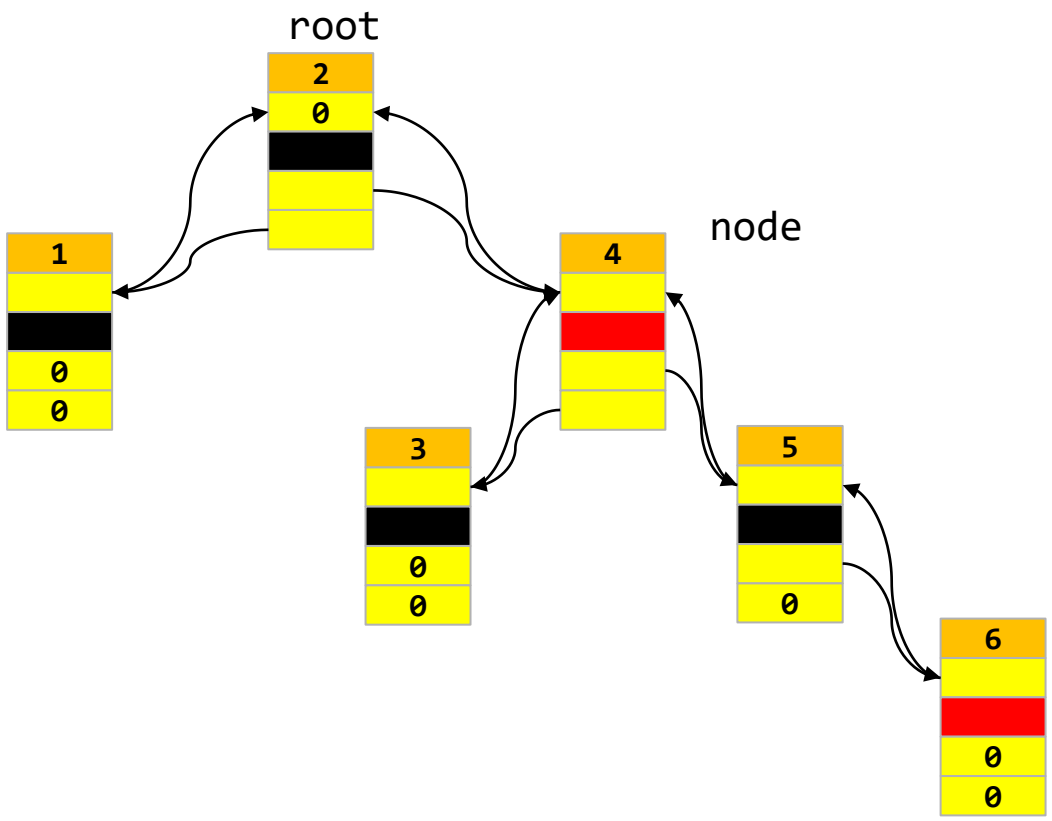




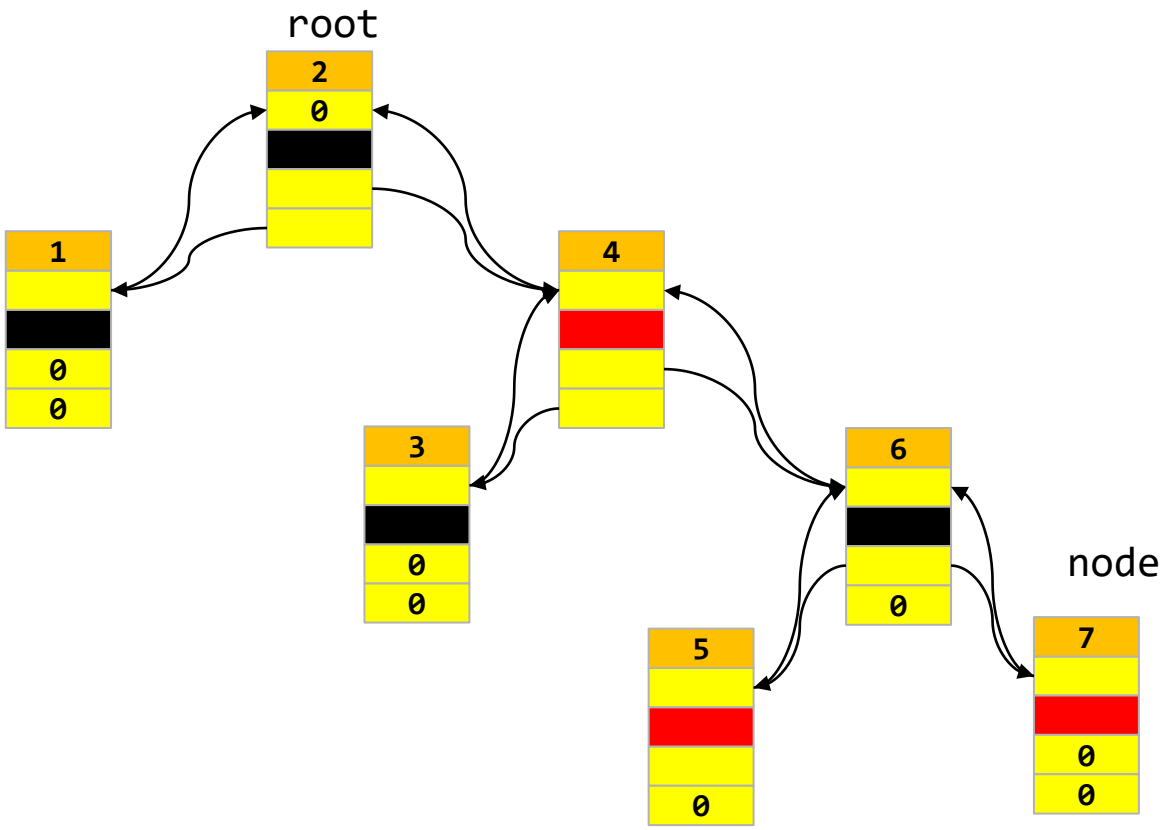
RB tree 삽입 코드 분석



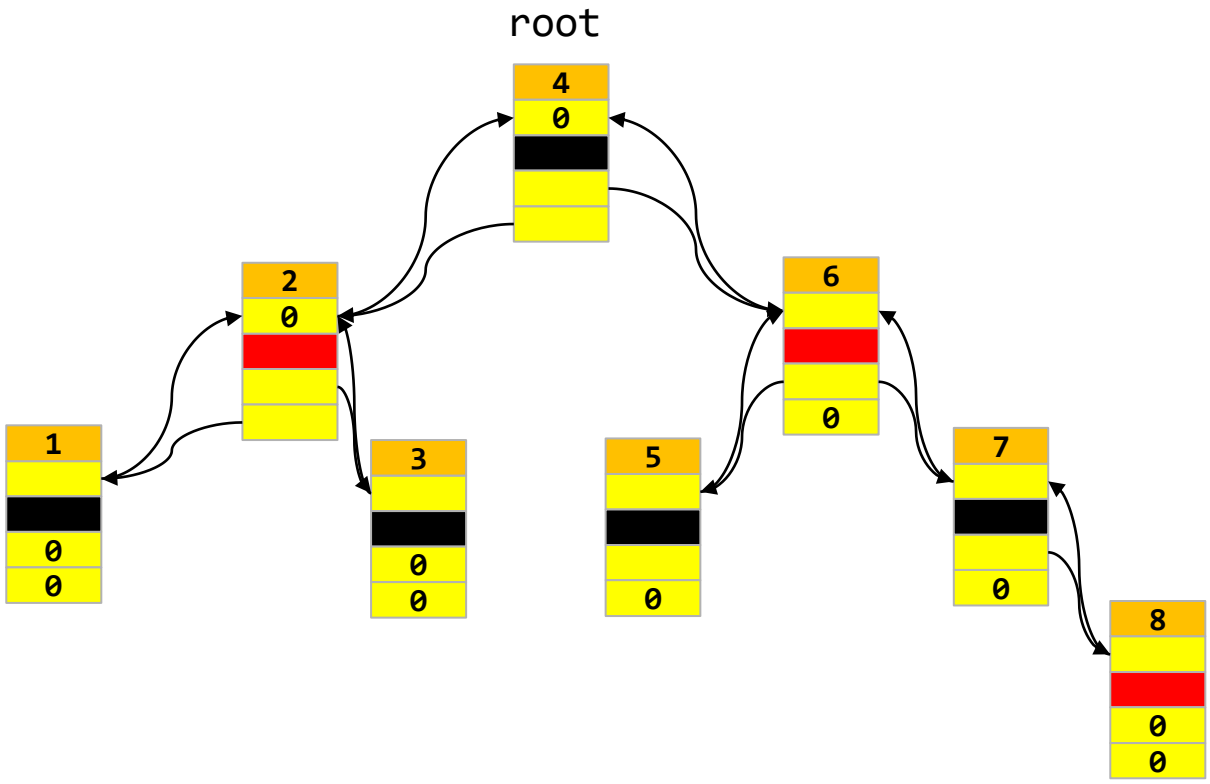
RB tree 삽입 코드 분석



RB tree 삽입 코드 분석

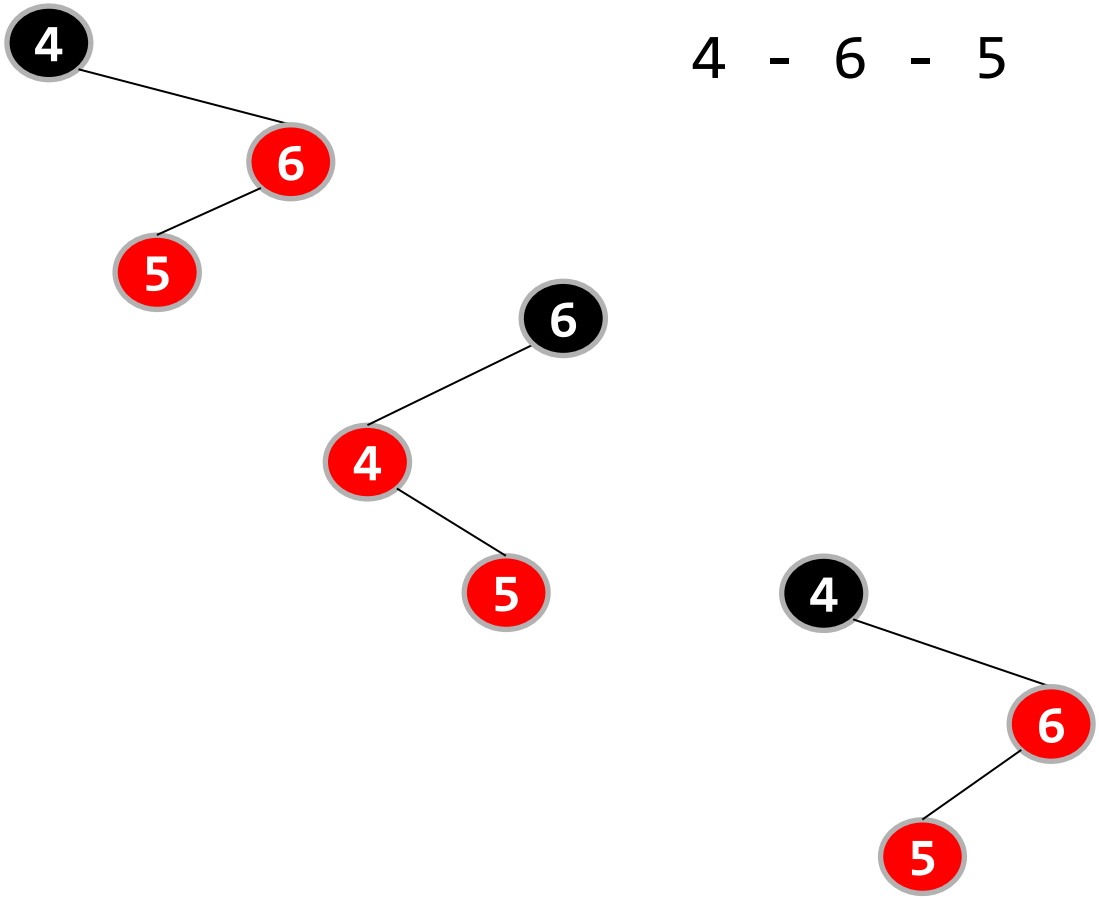


RB tree 삽입 코드 분석



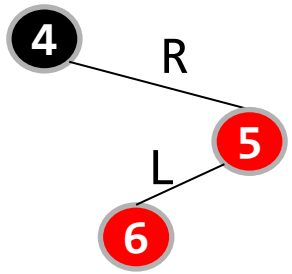
특수한 경사 처리 (R-L 경사)

L-R 경사나 R-L 경사는 한번의 회전으로는 밸런스를 잡을 수 없다.



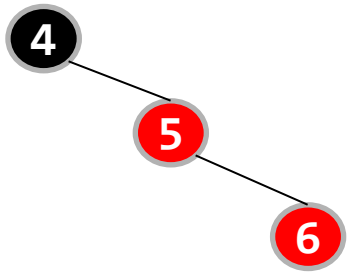
특수한 경사 처리 (R-L 경사)

R-L 경사인 경우는 처음에는 rotate_right(parent)로 들어온 순서를 바꾼후 rotate_left(gparent) 와 color flip으로 밸런스는 잡는다.

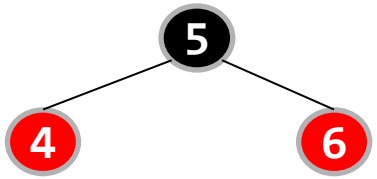


4 - 6 - 5
4 - 5 - 6
5 - 4 - 6

rotate_right(parent);

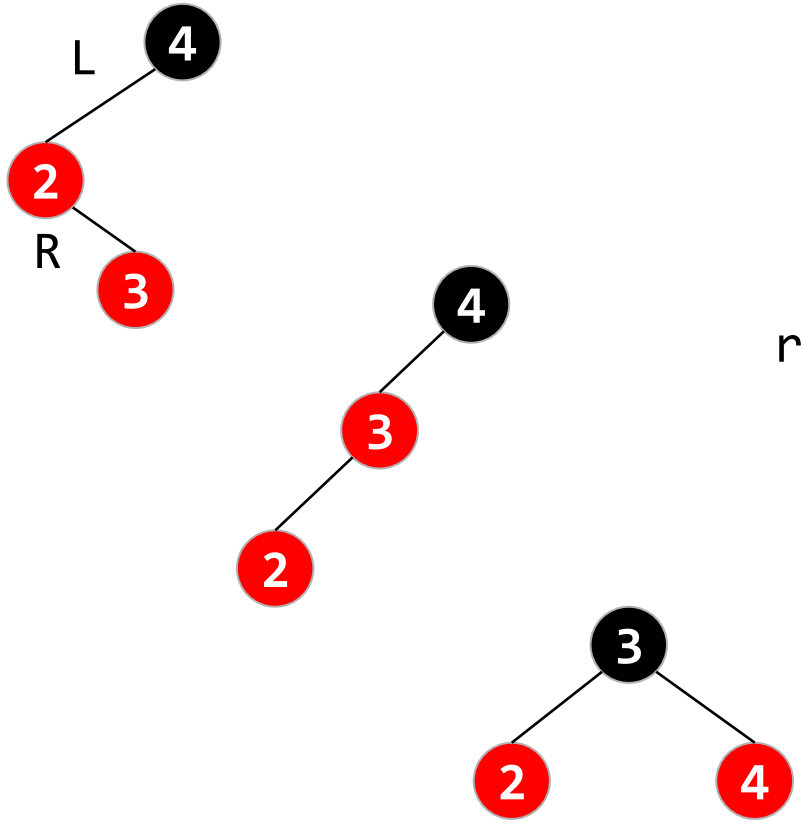


color flip
rotate_left(gparent);



특수한 경사 처리 (L-R 경사)

L-R 경사인 경우는 처음에는 rotate_left(parent)로 들어온 순서를
바꾼후 rotate_right(gparent) 와 color flip으로 밸런스는 잡는다.



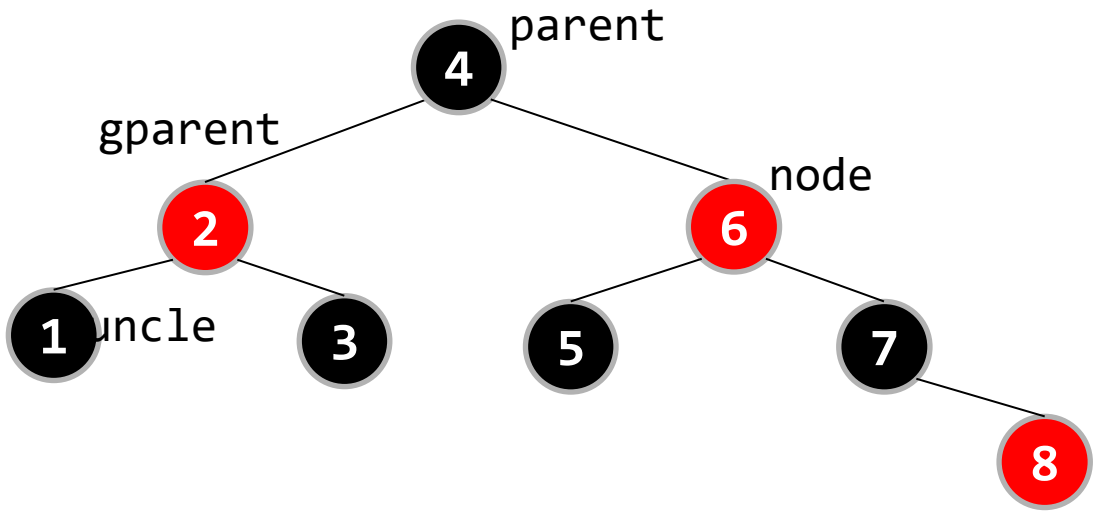
4 - 6 - 5
4 - 5 - 6
5 - 4 - 6

rotate_left(parent);

color flip
rotate_right(gparent);

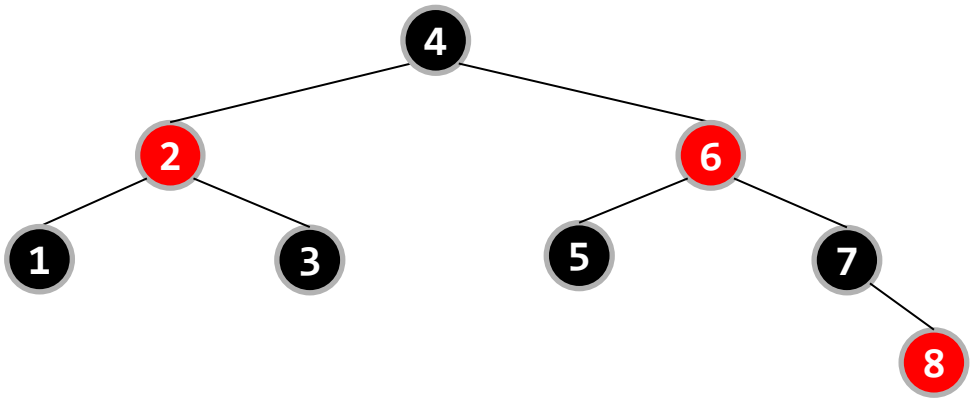
삽입완료 후 결과

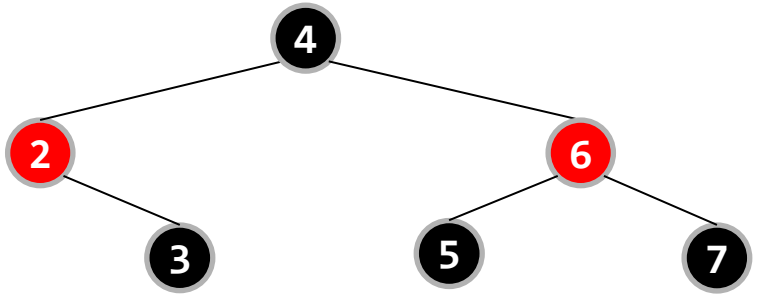
밸런스가 잡힌 후의 트리 모습 과 실제 실행 결과 비교



[4]
< 2>
< 6>
[1]
[3]
[5]
[7]
< 8>

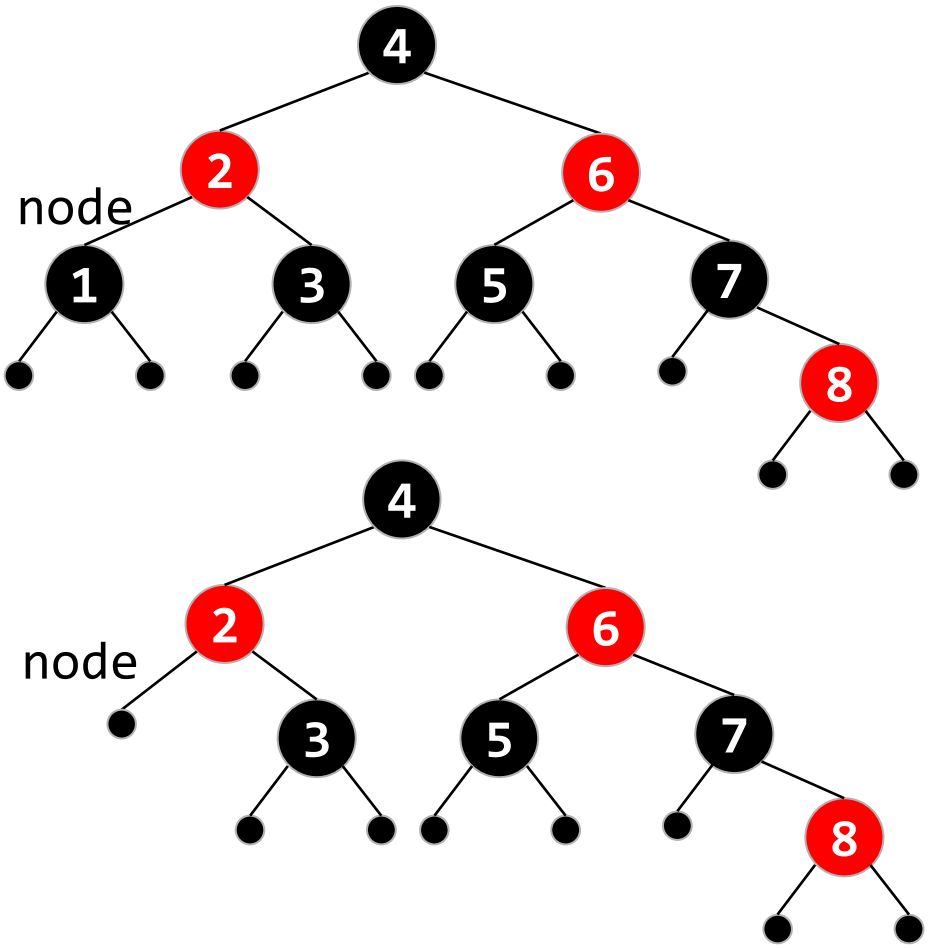
RB tree 의 삭제





만약 1을 지울 경우 ?
밸런스가 깨져 있다.

RB tree 의 삭제

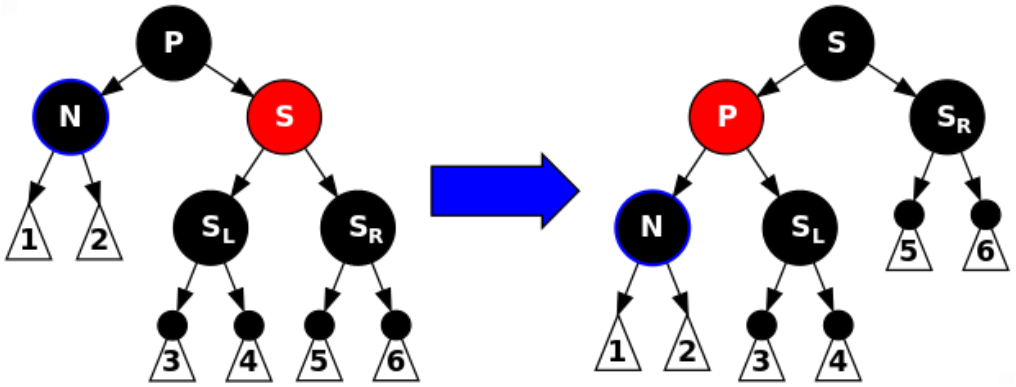


만약 1을 지울 경우 ?
밸런스가 깨져 있다.

NULL은 검정으로 간주 한다.

검정 노드를 지울 경우
밸런스가 깨진다.

RB tree 의 삭제



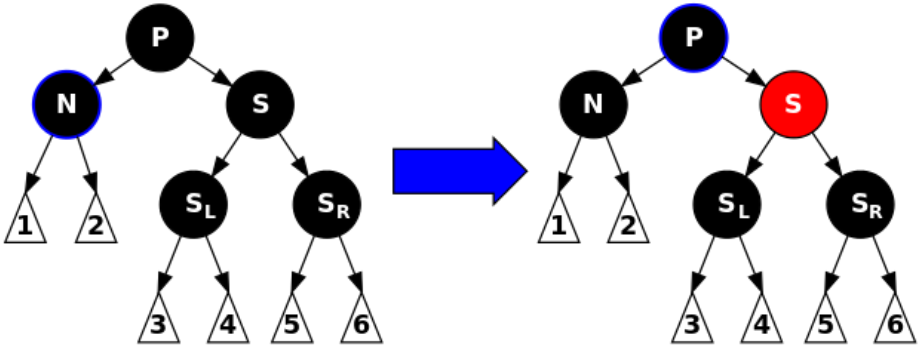
경우 1

형제 노드가 빨강인 경우

형제 노드가 오른쪽

```
color flip
    parent => RED
    형제   => BLACK
rotate_right(parent);
```

RB tree 의 삭제



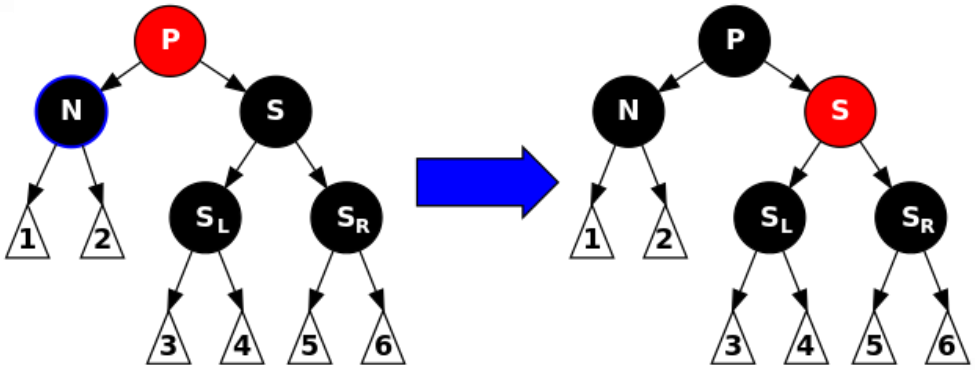
경우 2

부모, 형제, 모든 조카 => 검정

color flip

형제 => RED

RB tree 의 삭제

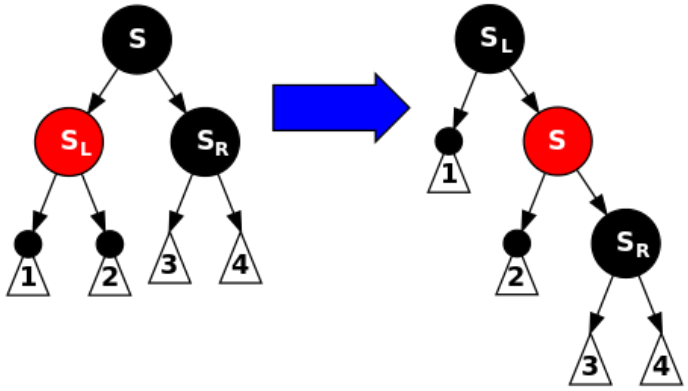


경우 3

모든 조카가 검정이고
부모가 빨강인 경우

color flip
  형제 => RED
  node => RED
  부모 => BLACK

RB tree 의 삭제



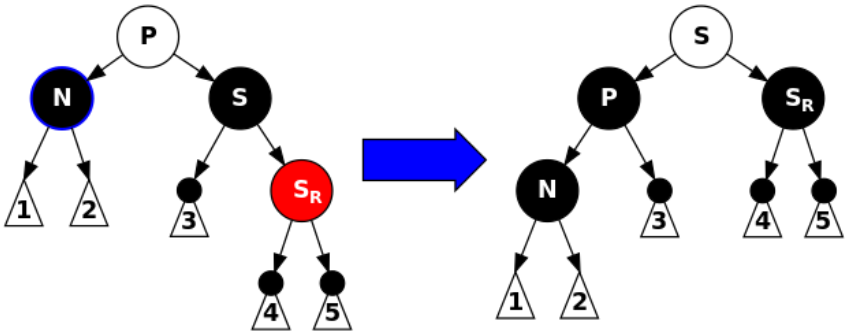
경우 4

가까운 위치의 조카가 빨강인 경우

color flip
    가까운 조카 => BLACK
    형제 => RED

rotate_right(s)

RB tree 의 삭제

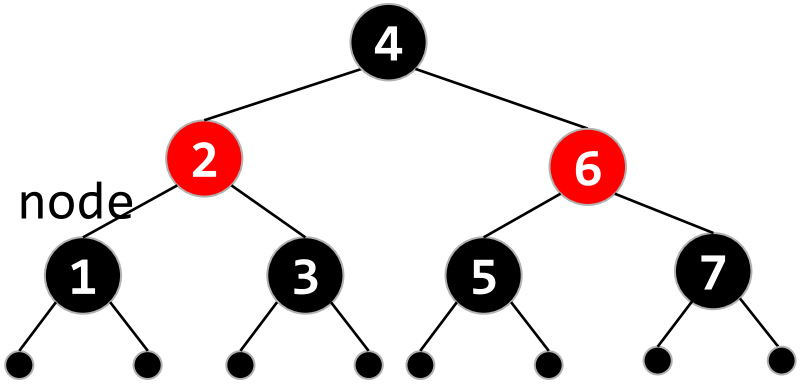


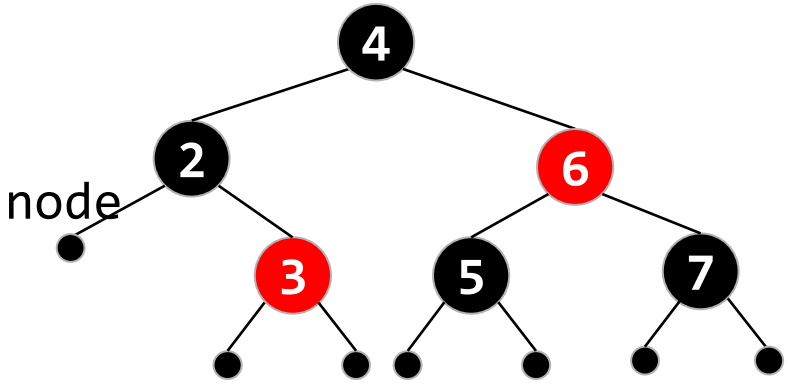
경우 5

위치가 먼 조카가 빨강인 경우

```
color flip
    parent => BLACK
    형제   => parent color
    먼 조카 => BLACK

rotate_left( parent );
```

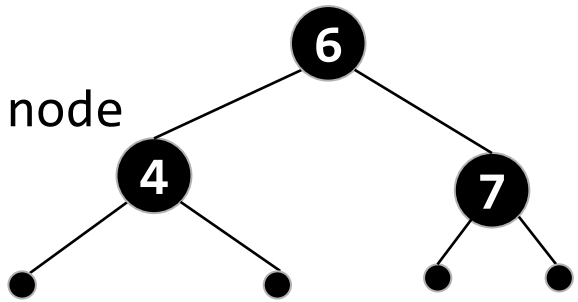
RB tree 의 삭제

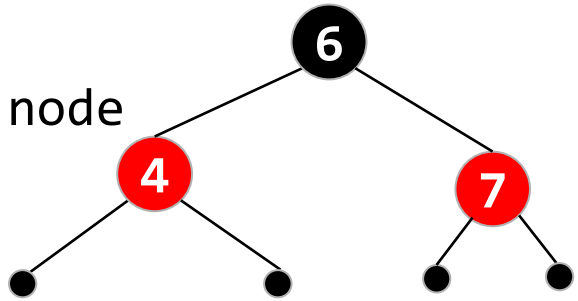


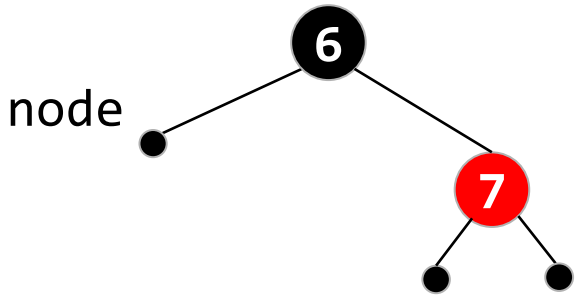


1을 지울 경우

RB tree 의 삭제

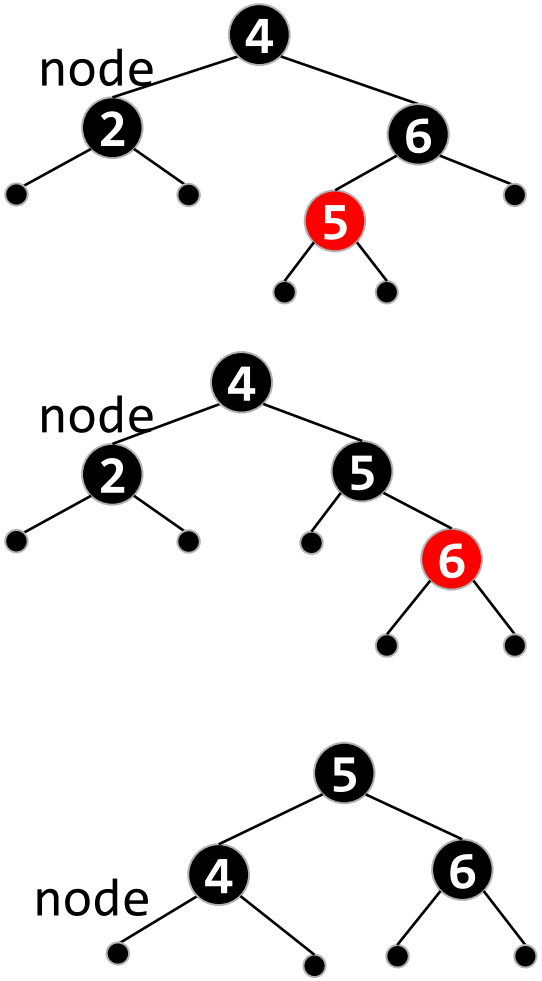






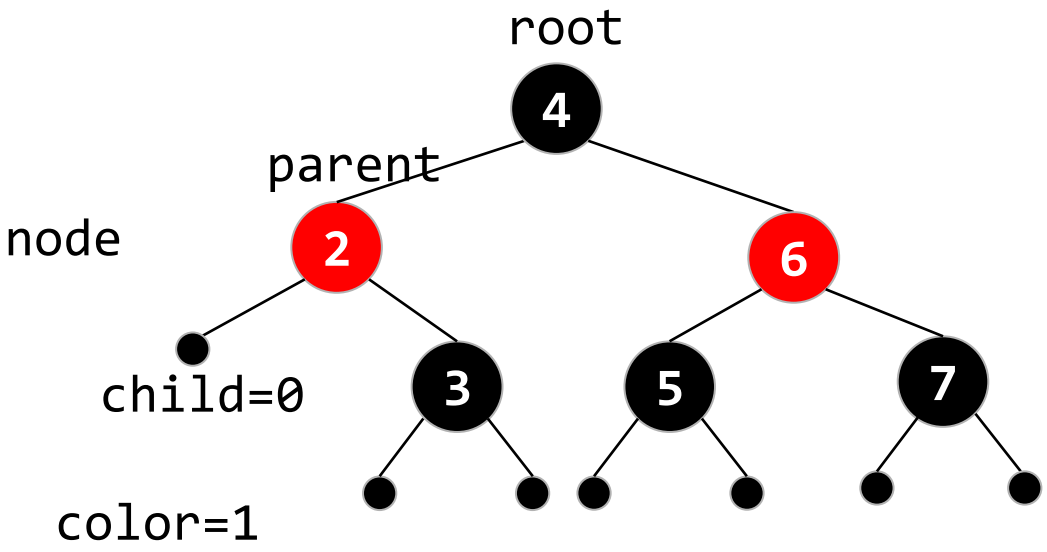
내릴 seed가 없으면
root로 부터 내린다.

RB tree 의 삭제

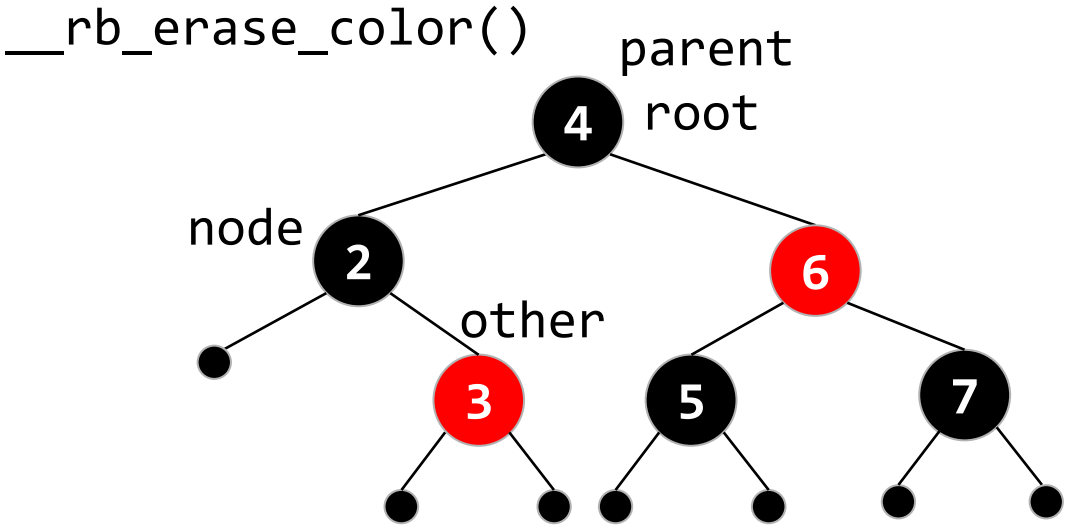


2를 지울 경우 :
가까운 조카가 빨강이므로
먼조카로 바꾼다음
특수한 회전을 통해서
밸런스를 잡는다.

RB tree 의 삭제 코드 분석

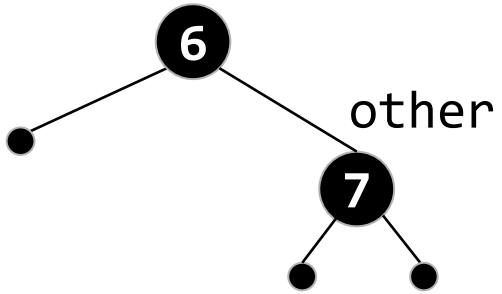


RB tree 의 삭제 코드 분석



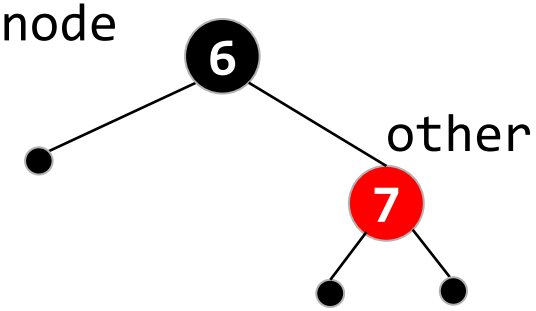
RB tree 의 삭제 코드 분석

color=1

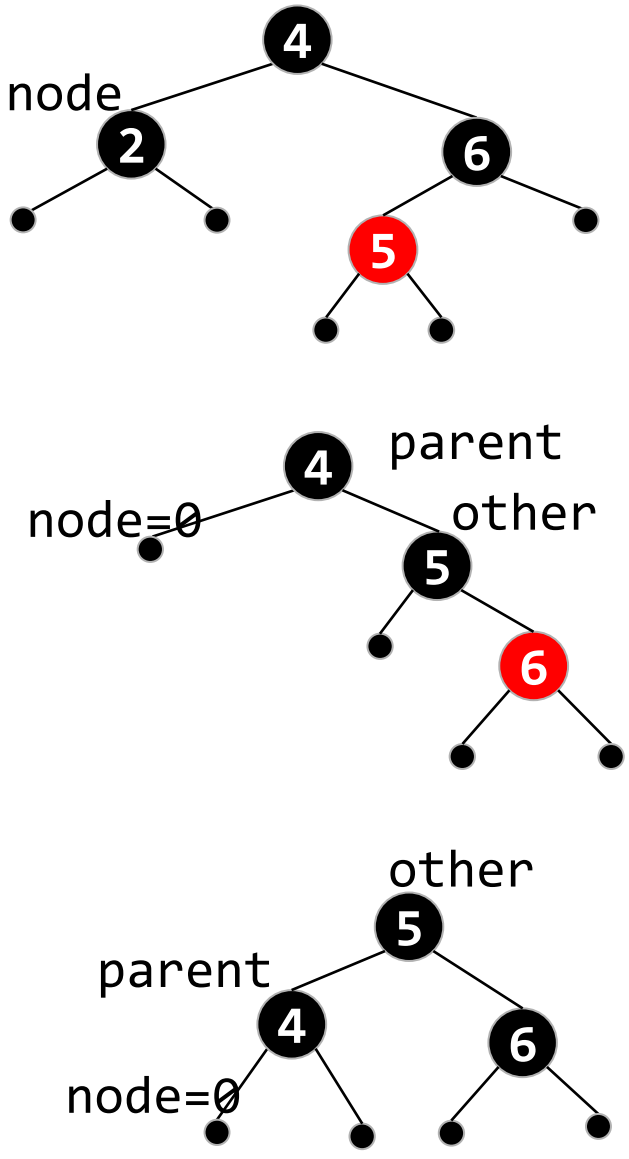


parent=0

node

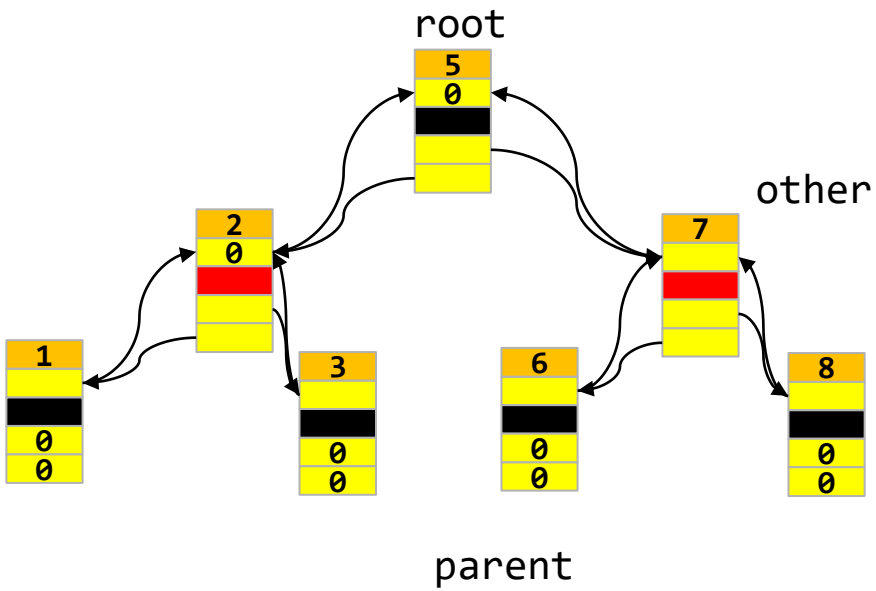


RB tree 의 삭제 코드 분석

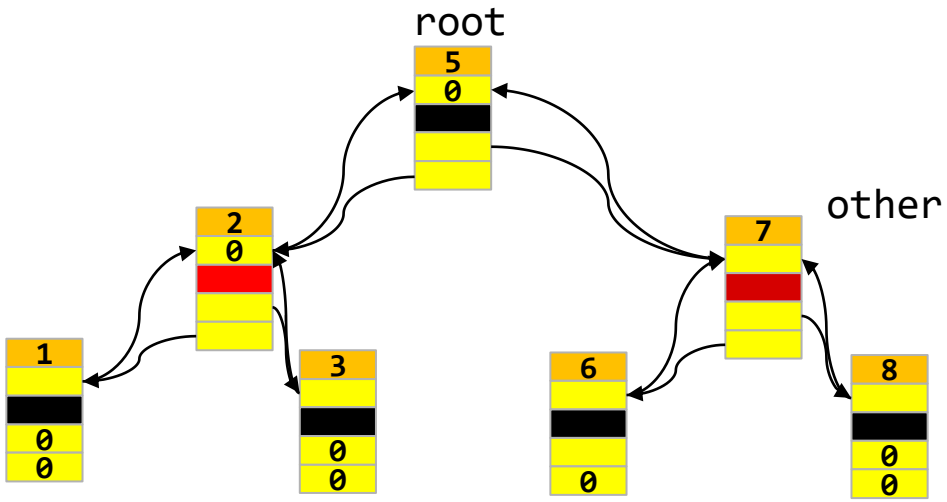


RB tree 의 삭제 코드 분석

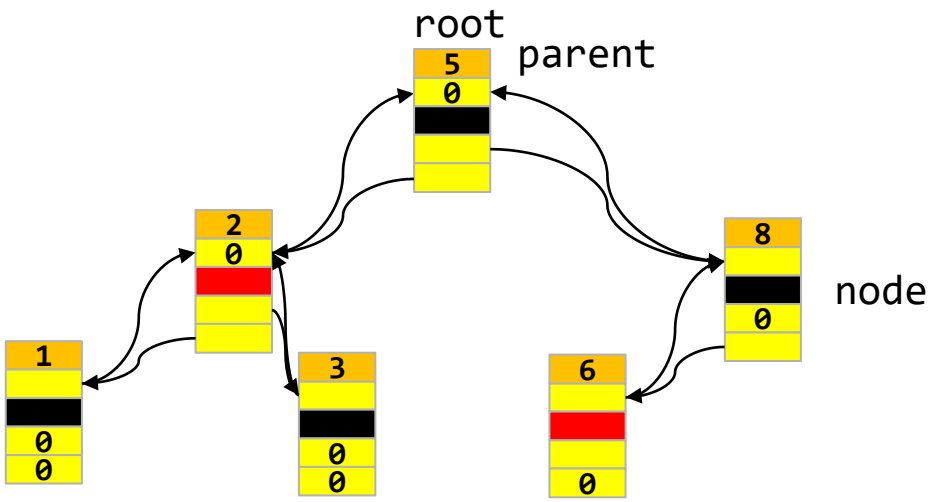
RB tree 에서는 중간 노드를 지울 경우
: 오른쪽 서브 트리중 가장 작은 값을
후보로 올린다.



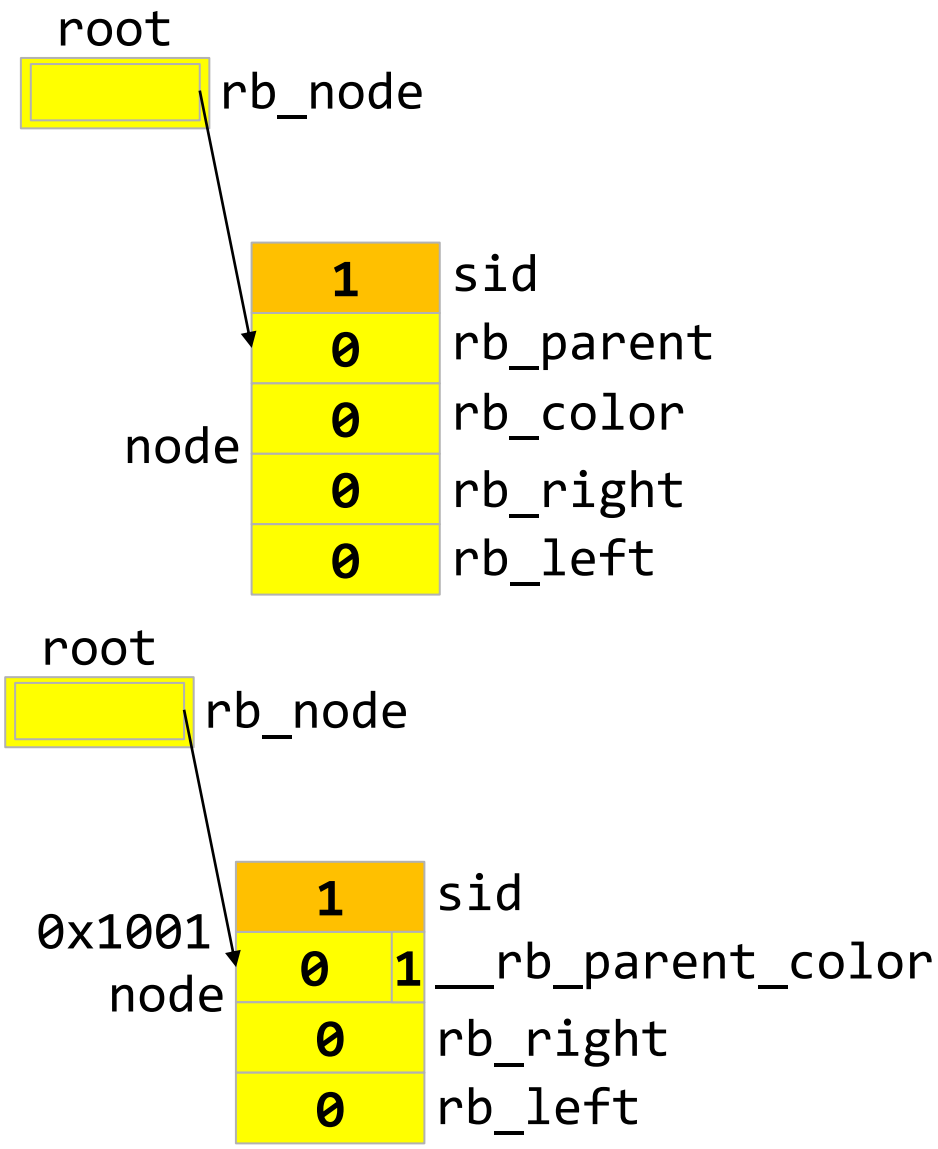
RB tree 의 삭제 코드 분석



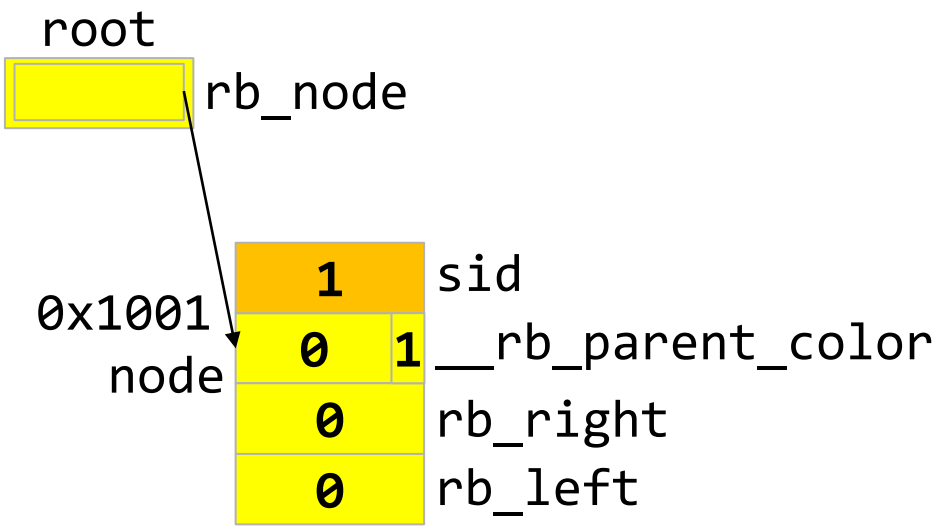
RB tree 의 삭제 코드 분석



RB tree 의 개선

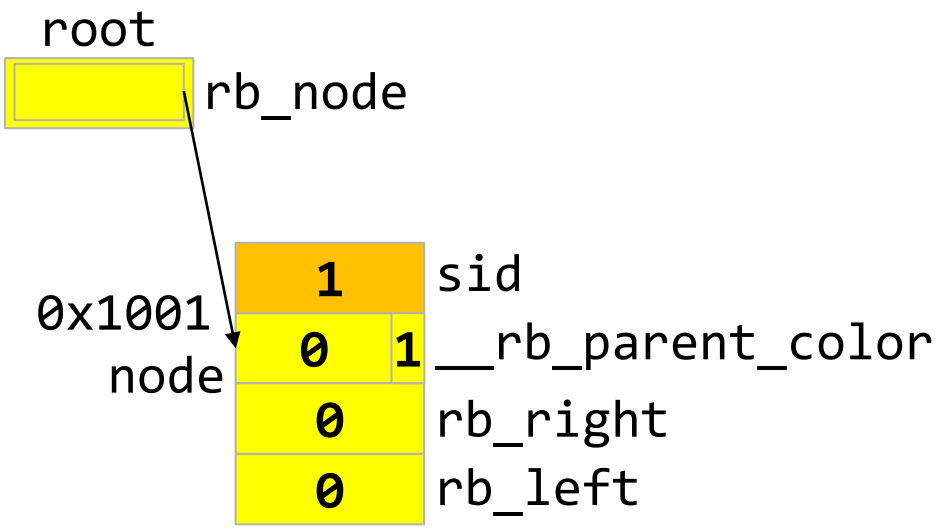

root
rb_node
1
sid
0
rb_parent
0
rb_color
node
0
rb_right
0
rb_left
root
rb_node
1
sid
0x1001
0
1
__rb_parent_color
node
0
rb_right
0
rb_left

RB tree 의 개선




```
rb_parent 매크로
0001000000000001
1111111111111100 &
--------------------
0001000000000000

 0x1000
```

RB tree 의 개선



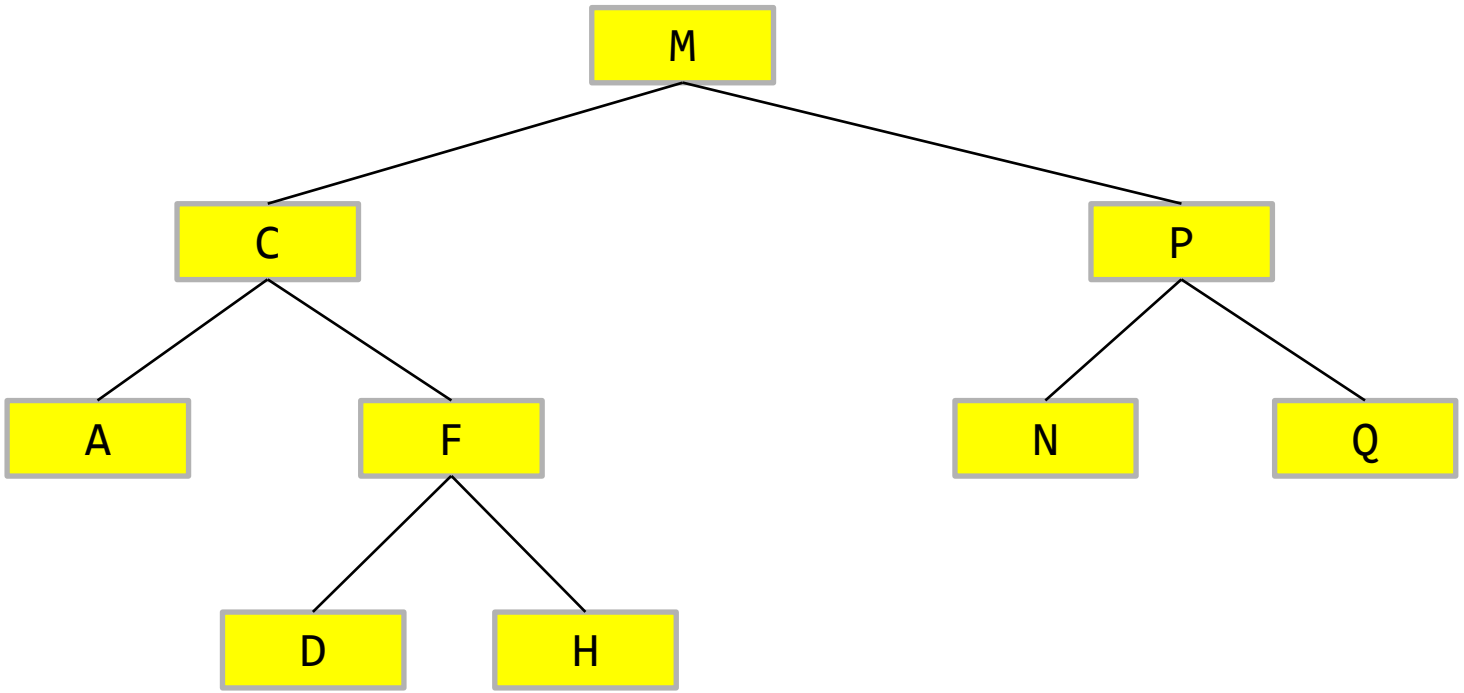

```
rb_color 매크로
0001000000000000
0000000000000001 &
--------------------
0000000000000000
```

증강 트리

알파벳 순서상 5번째 사람 ?

$O(\log_2 N)$

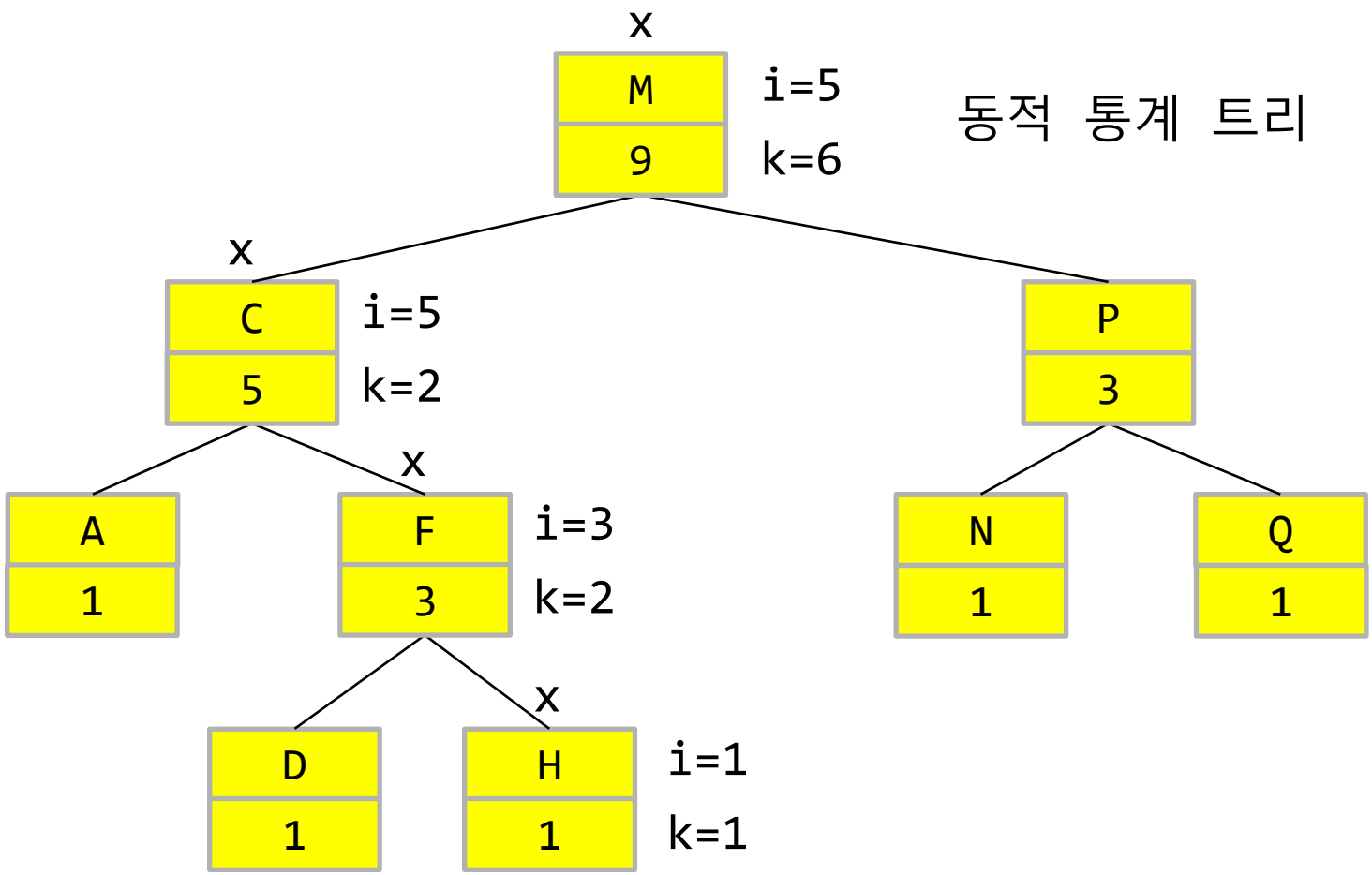
$O(N)$



증강 트리

알파벳 순서상 5번째 사람 ?

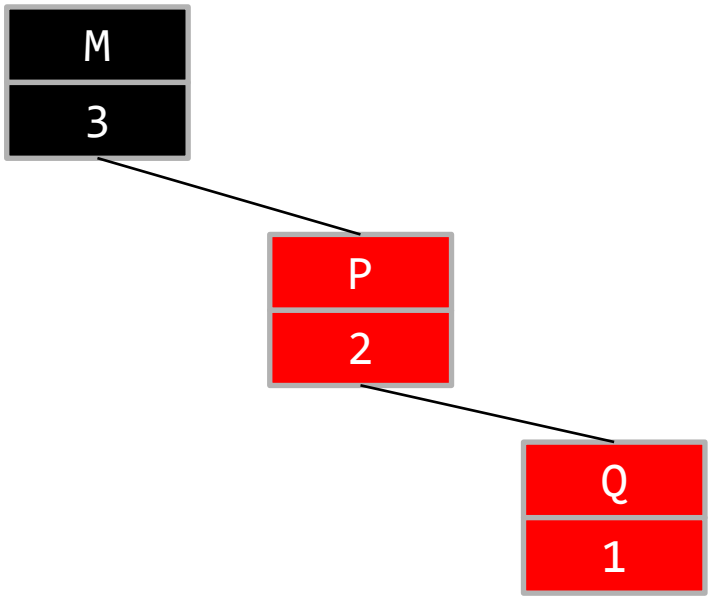
$O(\log_2 N)$

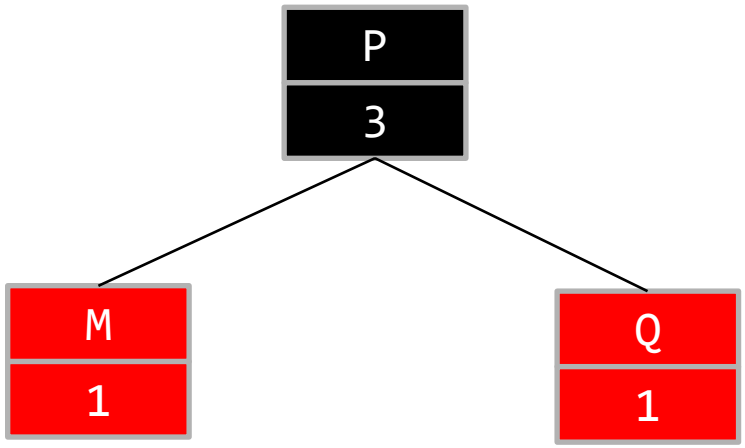


증강 트리

```
SAWON *my_select( struct rb_node *node, int i )
{
   int k=1;
   if(node == 0 )
      return 0;

   if( node->rb_left )
      k = rb_entry(node->rb_left,SAWON,node)->augmented+1;

   if( i==k )
      return rb_entry( node, SAWON, node );

   if( i<k )
     my_select( node->rb_left, i );
   else
     my_select( node->rb_right,i-k);
}
```

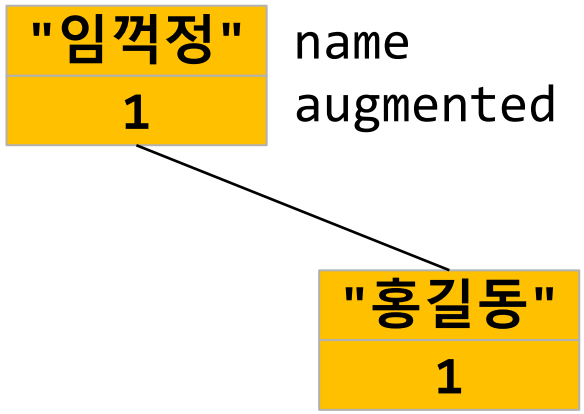
증강 트리



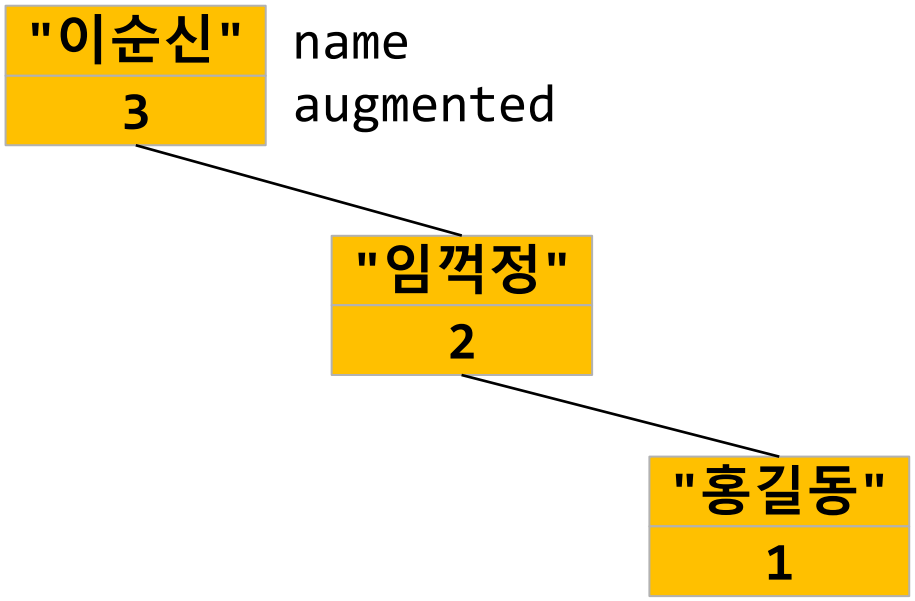



```
new->augmented = old->augmented;
old->augmented = recompute( old );
```

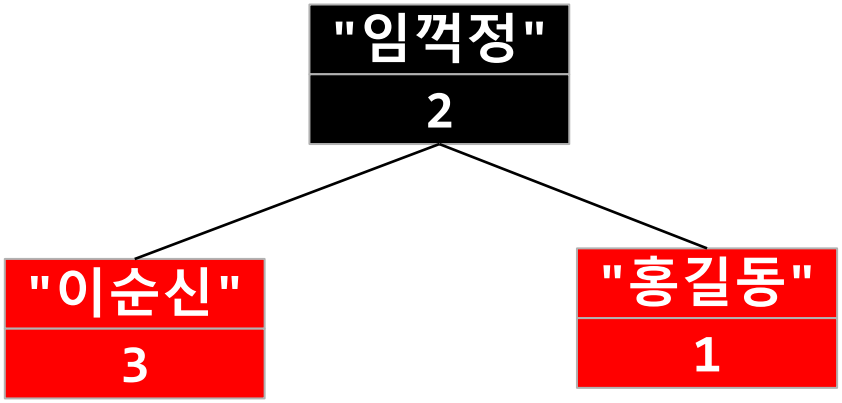
증강 트리



증강 트리



증강 트리

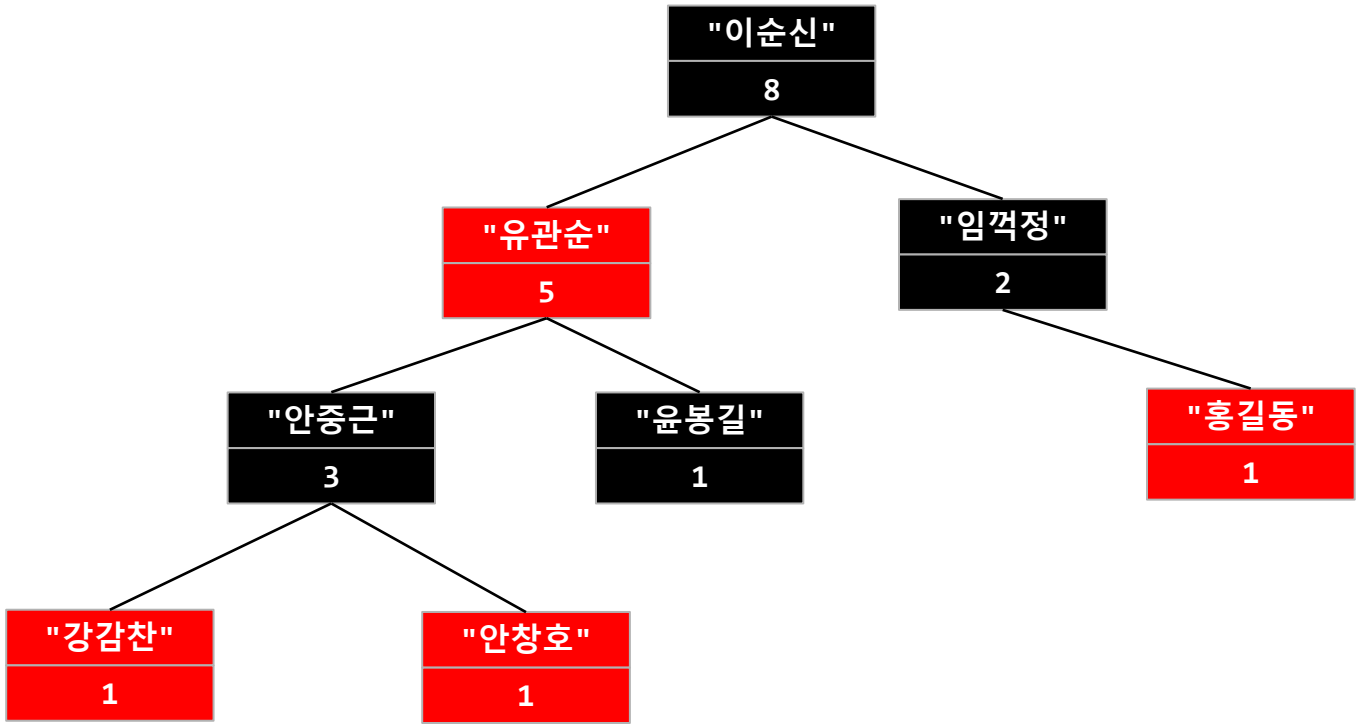


증강 트리

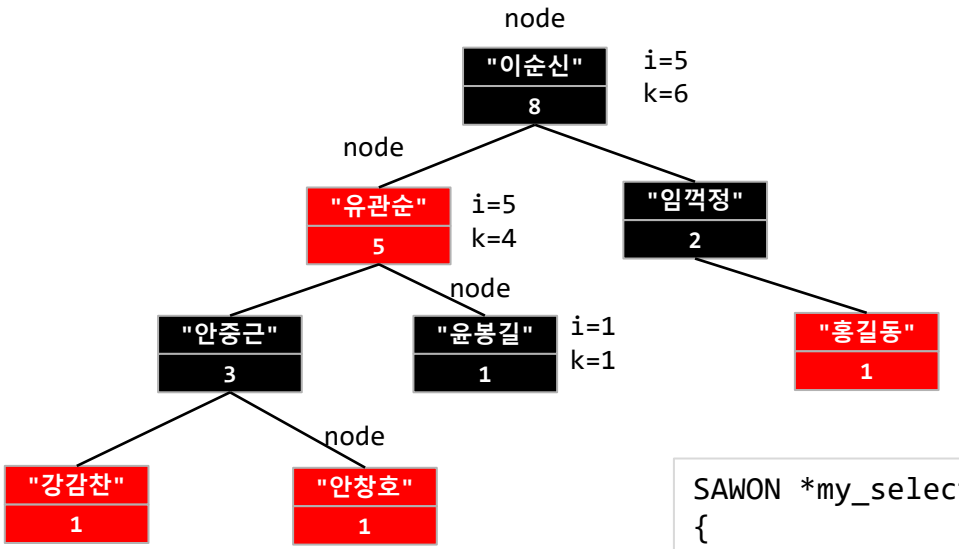
```
static void
augment_callbacks_rotate(struct rb_node *rb_old, struct rb_node *rb_new)
{
    SAWON *old = rb_entry(rb_old, SAWON, node);
    SAWON *new = rb_entry(rb_new, SAWON, node);
    new->augmented = old->augmented;
    old->augmented = my_compute(old);
}

RB_DECLARE_CALLBACKS( static, augment_callbacks, SAWON, node, int, augmented,
                      my_compute);
```

증강 트리

[이순신,8]
<유관순,5> [임꺽정,2]
[안중근,3] [윤봉길,1] <홍길동,1>
<강감찬,1> <안창호,1>

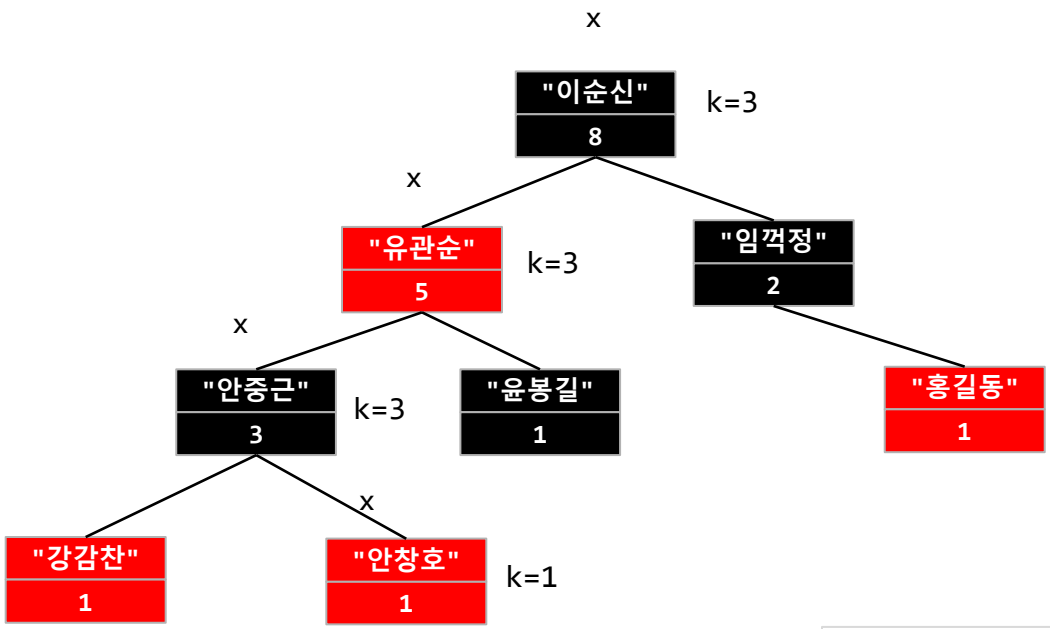


증강 트리



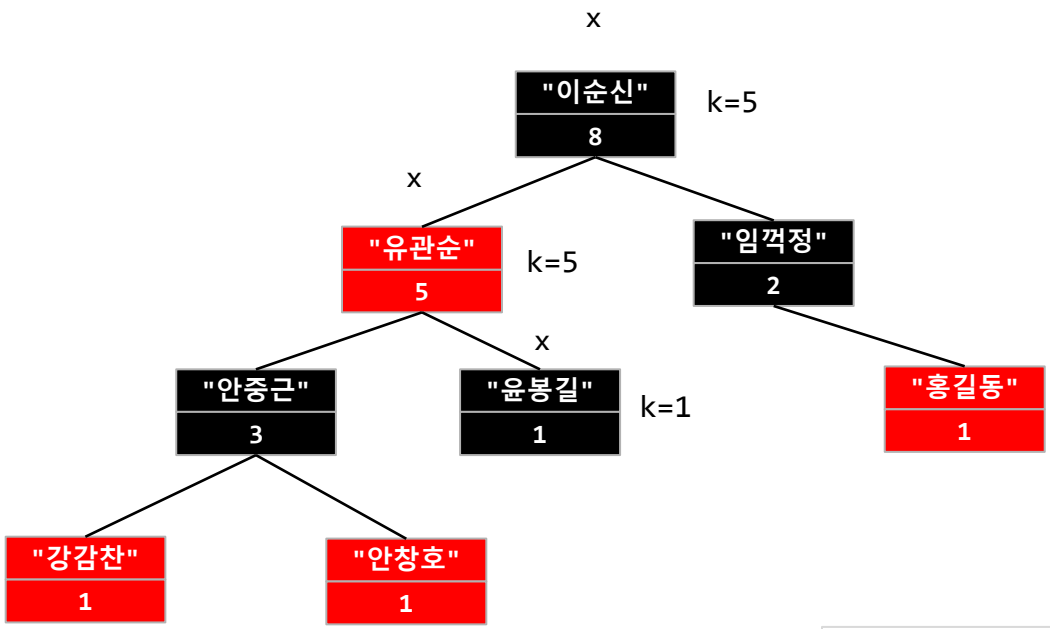

```
SAWON *my_select( struct rb_node *node, int i )
{
   int k=1;
   if(node == 0 )
      return 0;

   if( node->rb_left )
      k = rb_entry(node->rb_left,SAWON,node)->augmented+1;

   if( i==k )
      return rb_entry( node, SAWON, node );

   if( i<k )
     my_select( node->rb_left, i );
   else
     my_select( node->rb_right,i-k);
}
```

증강 트리



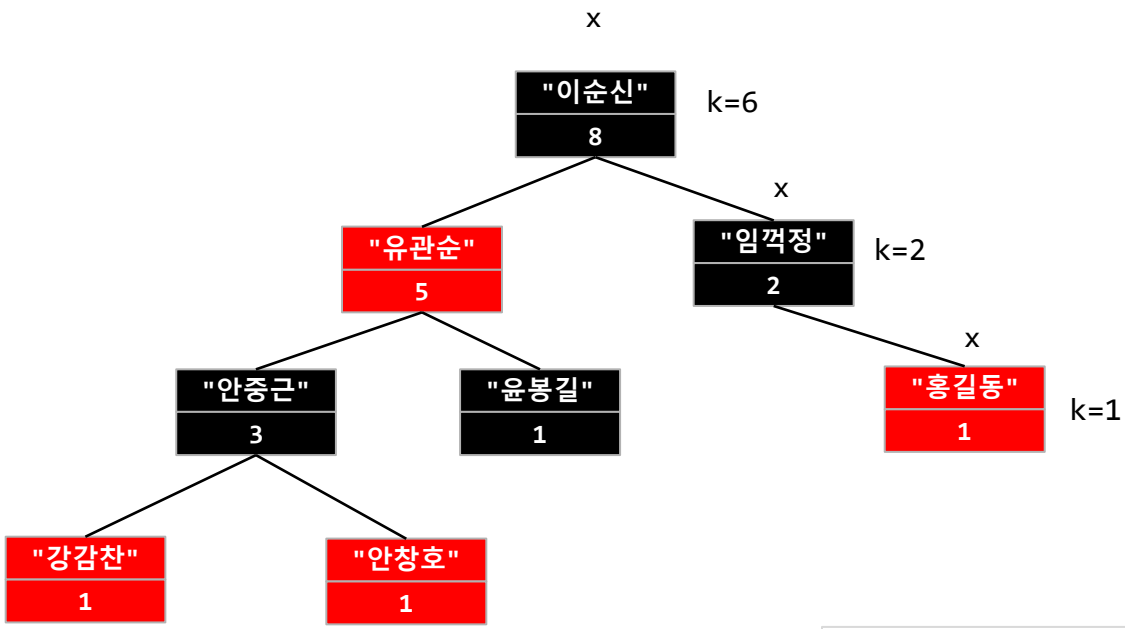

```
int my_rank( x )
{
   k <- size[left[x]]+1;

   while(x != 0)
   {
      if( x->parent->right == x )
        k+=size[x->parent->left]+1;
      x=x->parent;
   }
   return k;
}
```

증강 트리




```
int my_rank( x )
{
   k <- size[left[x]]+1;

   while(x != 0)
   {
      if( x->parent->right == x )
        k+=size[x->parent->left]+1;
      x=x->parent;
   }
   return k;
}
```

증강 트리

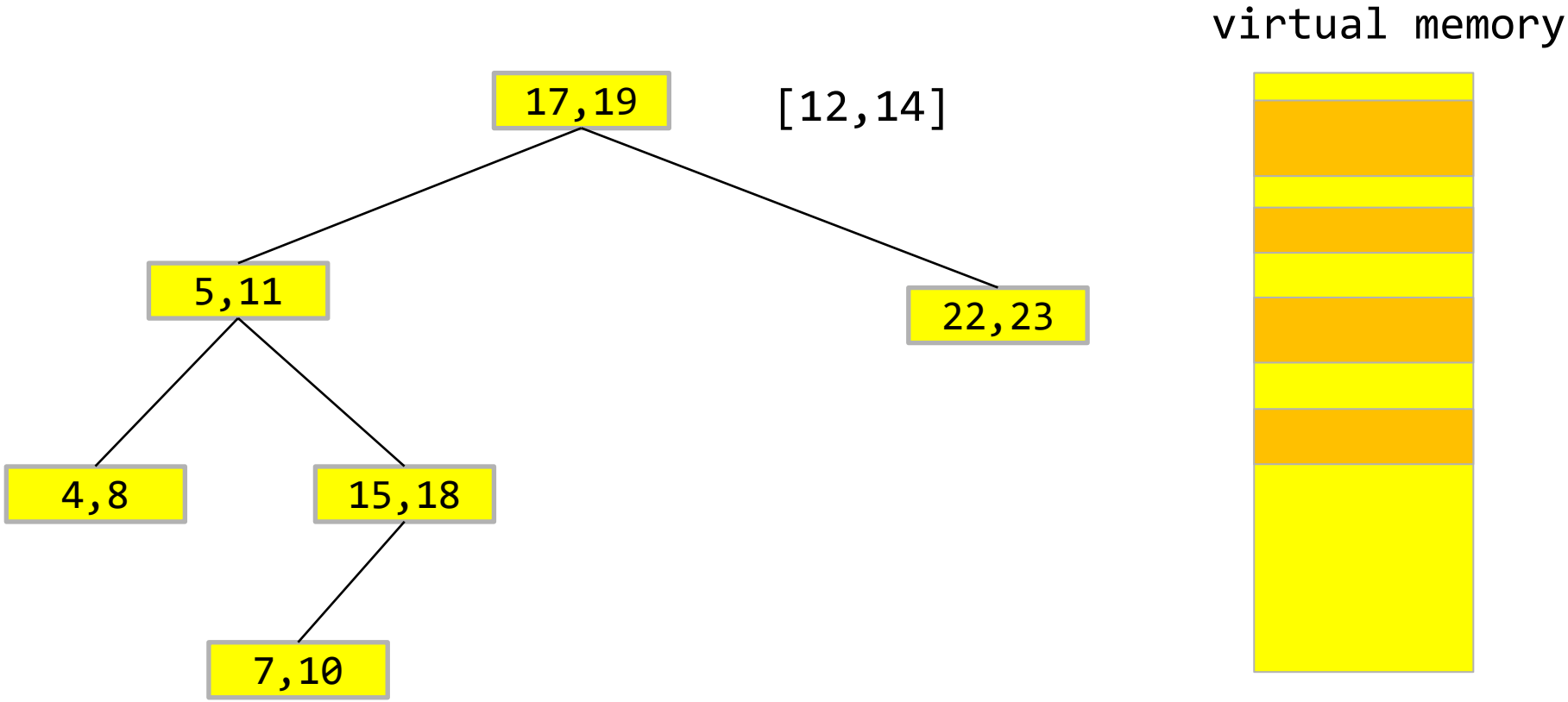



```
int my_rank( x )
{
   k <- size[left[x]]+1;

   while(x != 0)
   {
      if( x->parent->right == x )
        k+=size[x->parent->left]+1;
      x=x->parent;
   }
   return k;
}
```

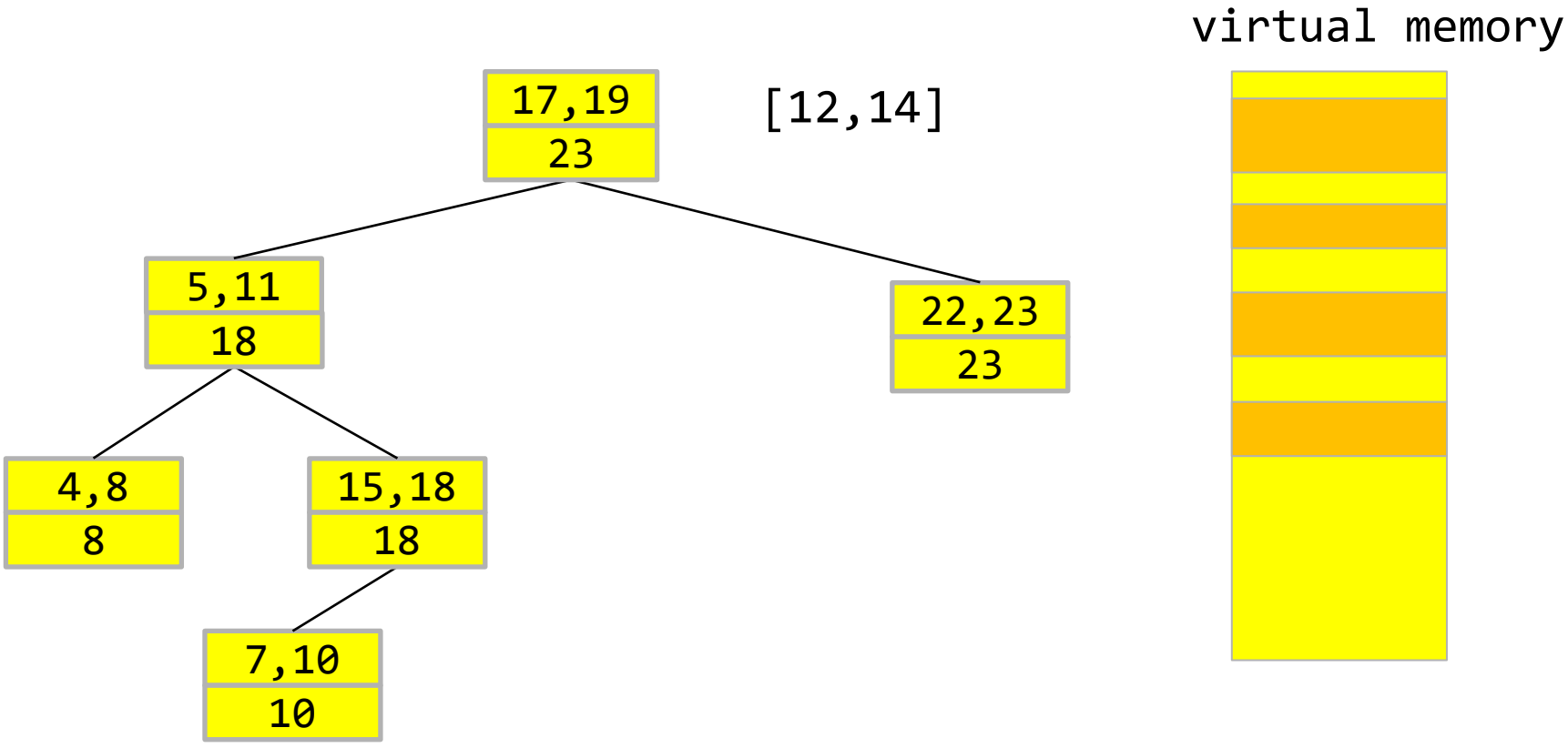
증강 트리

```
int my_rank( struct rb_node *node )
{
   int k=1;

   if( node->rb_left )
      k = rb_entry(node->rb_left, SAWON, node)->augmented+1;

   while(rb_parent(node) != 0)
   {
      if( rb_parent(node)->rb_right == node )
      {
        if(rb_parent(node)->rb_left )
          k+=rb_entry(rb_parent(node)->rb_left,SAWON,node)->augmented+1;
        else
          k++;
      }
      node=rb_parent(node);
   }
   return k;
}
```

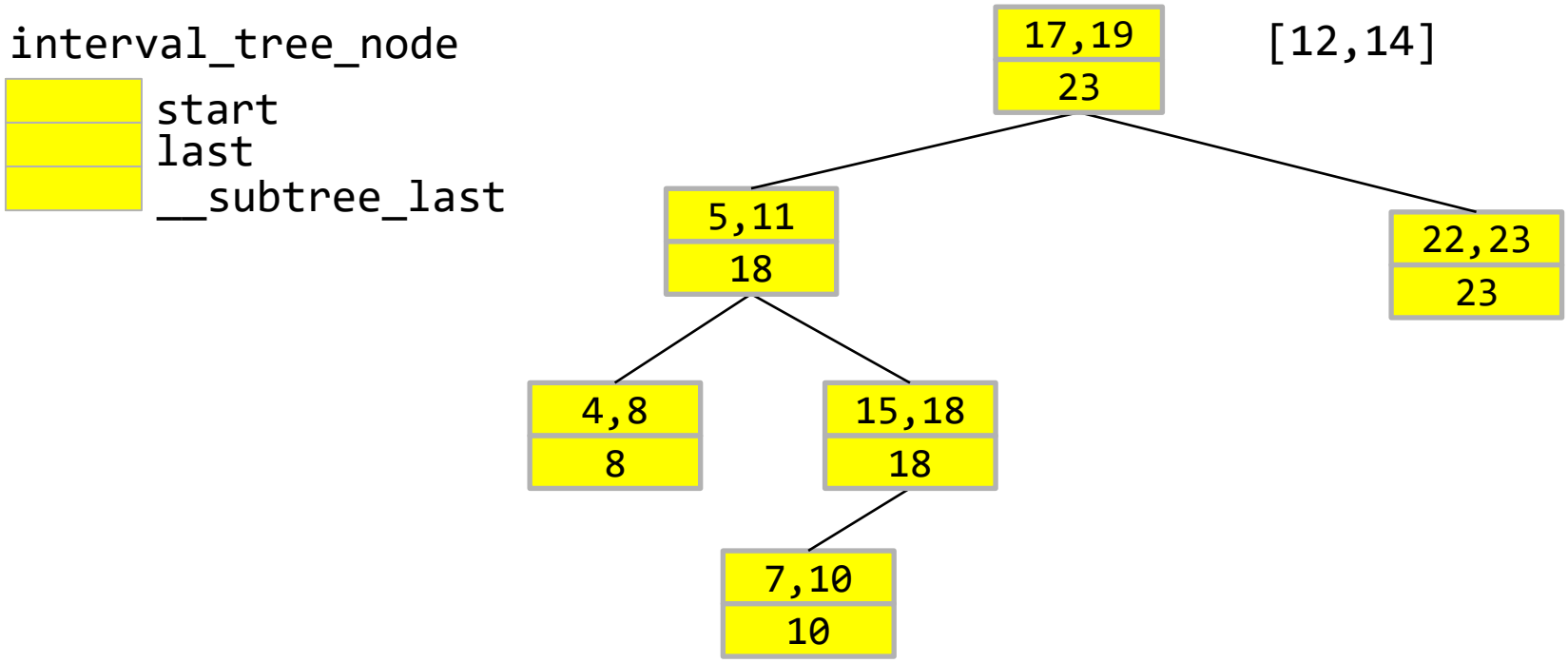
Interval 트리



Interval 트리



Interval 트리


interval_tree_node
start
last
__subtree_last
17,19
23
[12,14]
5,11
18
22,23
23
4,8
8
15,18
18
7,10
10

Interval 트리

```
INTERVAL_TREE_DEFINE(ITSTRUCT, ITRB, ITTYPE,
                     ITSUBTREE, ITSTART, ITLAST,
                     ITSTATIC, ITPREFIX)
INTERVAL_TREE_DEFINE(struct interval_tree_node,
                rb, unsigned long, __subtree_last,
                node->start, node->last, ,
                interval_tree)

unsigned long
interval_tree_compute_subtree_last
(struct interval_tree_node *node)
{
    unsigned long max = node->last;
    unsigned long subtree_last;
    if( node->rb.rb_left )
    {
      subtree_last = rb_left->__subtree_last;
      if( max < subtree_last )
        max = subtree_last;
    }
}
```

Interval 트리

```
unsigned long
interval_tree_compute_subtree_last
(struct interval_tree_node *node)
{
    unsigned long max = node->last;
    unsigned long subtree_last;
    if( node->rb.rb_left )
    {
      subtree_last = rb_left->__subtree_last;
      if( max < subtree_last )
        max = subtree_last;
    }

    if( node->rb.rb_right )
    {
      subtree_last = rb_right->__subtree_last;
      if( max < subtree_last )
        max = subtree_last;
    }
    return max;
}
```

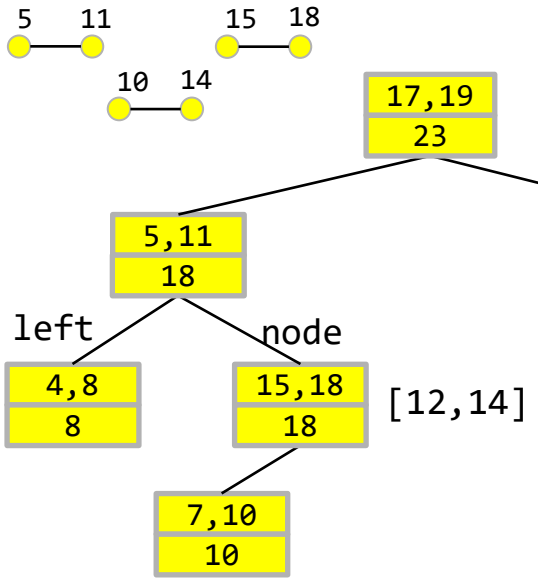
Interval 트리

```
void interval_tree_insert(struct interval_tree_node *node,
                          struct rb_root *root)
{
    struct rb_node **link = &root->rb_node, *rb_parent = NULL;
    unsigned long start = node->start, last = node->last;
    struct interval_tree_node *parent;

    while (*link) {
        rb_parent = *link;
        parent = rb_entry(rb_parent, struct interval_tree_node
                          , rb);
        if (parent->__subtree_last < last)
            parent->__subtree_last  = last;
        if (start < parent->start)
            link = &parent->rb.rb_left;
        else
            link = &parent->rb.rb_right;
    }
    rb_link_node(&node->rb, rb_parent, link);
    rb_insert_augmented(&node->rb, root,
                 &interval_tree_augment);
}
```

Interval 트리

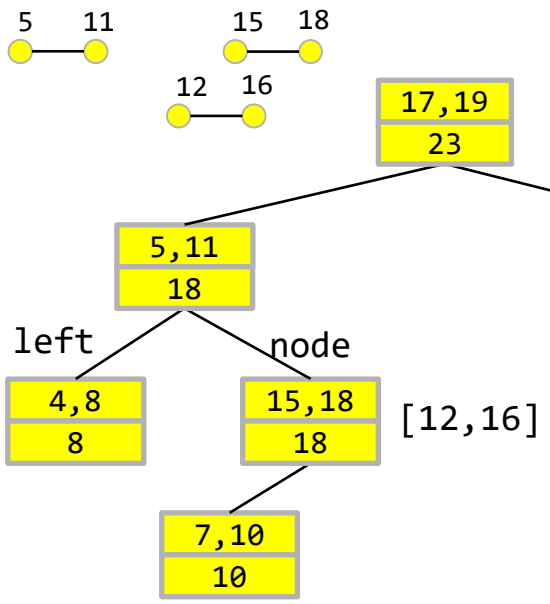
```
struct interval_tree_node *
interval_tree_subtree_search(
struct interval_tree_node *node,
unsigned long start,
unsigned long last)
{
    while (true) {
        if (node->rb.rb_left) {
             struct interval_tree_node *left = node->rb.rb_left;
             if (start <= left->__subtree_last) {
                node = left;
                continue;
             }
        }
        if (node->start <= last) {
            if (start <= node->last )
                return node;
            if (node->rb.rb_right) {
                node = rb_entry(node->ITRB.rb_right, ITSTRUCT, ITRB);
                if (start <= node->__sub_tree_last)
                    continue;
            }
        }
        return NULL;
    }
}
```

Interval 트리




```
struct interval_tree_node *
interval_tree_subtree_search(struct interval_tree_node *node,
unsigned long start,
unsigned long last)
{
    while (true) {
        if (node->rb.rb_left) {
             struct interval_tree_node *left = node->rb.rb_left;
             if (start <= left->__subtree_last) {
                node = left;
                continue;
             }
        }
        if (node->start <= last) {
            if (start <= node->last )
                return node;
            if (node->rb.rb_right) {
                node = rb_entry(node->ITRB.rb_right, ITSTRUCT, ITRB);
                if (start <= node->__sub_tree_last)
                    continue;
            }
        }
        return NULL;
    }
}
```

Interval 트리




```
struct interval_tree_node *
interval_tree_subtree_search(struct interval_tree_node *node,
unsigned long start,
unsigned long last)
{
    while (true) {
        if (node->rb.rb_left) {
             struct interval_tree_node *left = node->rb.rb_left;
             if (start <= left->__subtree_last) {
                node = left;
                continue;
             }
        }
        if (node->start <= last) {
            if (start <= node->last )
                return node;
            if (node->rb.rb_right) {
                node = rb_entry(node->ITRB.rb_right, ITSTRUCT, ITRB);
                if (start <= node->__sub_tree_last)
                    continue;
            }
        }
        return NULL;
    }
}
```